INDENICA Variability Implementation Language: Language Specification

Version 0.91

(corresponds to VIL bundle versions 0.4.0 and VTL bundle version 0.4.0)

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

Creating domain-specific service platforms requires the capability of customizing and configuring service platforms according to the specific needs of a domain. In this document we provide a novel approach for variability implementation. We focus on how to implement selected customization and configuration options in service (platform) ecosystems in a generic way focusing on the specification of the instantiation process. This enables domain engineers to define their specific instantiation process in a declarative way without the need for implementation of specific tool components such as instantiators.

In this document we specify the concepts of the INDENICA variability implementation language (VIL) for specifying how customization and configuration options in service (platform) ecosystems can be turned into (instantiated) artefacts.

Version

0.5	15. June 2013	first version derived from D2.2.2
0.6	8. August 2013	revised concepts
0.7	26. September 2013	revisions based on actual implementation
0.8	21. October 2013	Jars and zips added, clarifications for ‘map’, RHSMATCH and further built-in operations, qualified names, script parameter sequence refined, pattern path vs. artefact creation clarified, ITER clarified, further XML operations
0.85	30. October 2013	Starting the “How to ...?” section, classloader registration for Java Extensions in VTL
0.86	14. November 2013	Additional XML operations such as constructors, SCRIPTDIR VIL variable
0.87	12. December 2013	Refined hierarchical import path, named-base access to VTL scripts (in addition to file-path based access), final separator expression in for-loop, toJavaPath
0.87	19. December 2013	Java artefacts
0.88	10. February 2014	Project predecessors and further project operations, instantiation via scripts in other projects (instantiate command)
0.89	19. March 2014	Revision of automatic conversions (most specific one is applied).
0.90	04. April 2014	Versions for VTL in VIL scripts.
0.91	21. April 2014	Implementation status, detail fixes, renamePackages in Java Artifact

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Figures

1 Introduction

This document specifies the INDENICA variability implementation language (VIL) in terms of a living document, which describes the most current version of the language based on discussions with the partners and experiences made during the project.

VIL consists two languages: a build process description language and a template language. The focus of the VIL build language is on instantiating a whole service platform in terms of a software product line, while the template language focuses on the instantiation and creation of individual artefacts. Both VIL languages are based on an explicit and extensible artefact model as well as a tight integration with the INDENICA variability modelling language IVML [3].

The remainder of this language specification is structured as follows: in Section 2, we will briefly introduce the VIL approach. In Section 3, we will define the syntax and semantics of the aforementioned VIL languages, their common expression language as well as the underlying type system (including the artefact model and the IVML integration). Finally, in Section 4, we will provide the grammars of the VIL languages as a reference.

2 The INDENICA Variability Implementation Approach

In this section, we describe the concepts of the INDENICA Variability Implementation Language (VIL). VIL is designed to realize the instantiation of artefacts in a generic way, i.e., using a specification-based approach instead of relying on domain- or product-line-specific implemented instantiation mechanisms. A more detailed discussion of the approach idea and its benefits for product line engineering can be found in D2.2.2 [5].

The VIL is more than a single language. It consists of two languages and requires the understanding of additional core concepts:

- **Artefact meta-model:** Everything that can be instantiated (transformed or generated) is regarded as an artefact. The VIL approach relies on an artefact meta-model as its foundation. The artefact meta-model (or often artefact model for short) describes what operations can be performed on certain types of artefacts, such as Java source code, Java byte code, XML files but also components (for runtime variabilities), or elements of the file system such as files or folders. Production strategies are operations on the types of the input and output artefacts using the capabilities of the assets for specifying the instantiation.
- **VIL template language** is used to instantiate a certain type of target artefact in a reusable way. Basically, the VIL template language covers generation as well as transformation-based production strategies.
- **Blackbox instantiators:** In some situations it might be difficult, inconvenient, or even impossible to describe a production strategy using the VIL template language. One example is the Cocktail instantiator discussed in Deliverable D2.2.2 [5] as it mainly modifies Java bytecode and, thus, it is easier to realize (at least some part of it) in an usual programming language such as Java, i.e., from the point of view of VIL as a black box. Another example is a programming language compiler or a linker, which should not be re-developed using VIL but simply reused. In case of legacy product lines, an existing instantiator may be called or wrapped into a VIL extension.
- **VIL build language:** This is the main part of the VIL language as it binds all other pieces together. This is used to define individual production strategies, i.e., to relate artefacts and instantiation mechanisms, to combine production strategies in terms of rules and to specify the execution of the rules. Basically, it is a rule-based programming language as a foundation for describing product line instantiation processes.

VIL and its sublanguages are tightly integrated with IVML, i.e., IVML identifiers and configuration values can be directly used in VIL. From a more general point of view, VIL and its sublanguages rely on existing, practically proven concepts such as build rules or template languages in order to avoid reinventing the wheel. However, existing concepts as well as related tooling does not provide the full support for variability instantiation as we experienced in our analysis of related technologies.

Thus, we reuse and extend existing concepts to apply it to variability realization and created the VIL as a completely new language along with a novel implementation.

3 The VIL Languages

In this section, we will describe the two (sub) languages of VIL, i.e., the VIL build language and the VIL template language, as well as their main concepts. Due to the nature of both languages as variability implementation languages, they share a common type system as well as a common expression language.

The **extensible VIL type system** is the foundation for both VIL sub languages. The type system consists of basic types such as Integer or Boolean, configuration-related types realizing the integration with an IVML [3] variability model, artefact-related types implementing the artefact (meta) model, implicit types representing instantiators and derived types such as containers. In particular, the type system is extensible, i.e., additional or refining artefact types or instantiators can easily be added (in terms of Java classes). If compared with an object-oriented language, the artefact types can be considered as classes, the operations as methods, individual artefacts as instances and the execution of artefact operations as method calls. However, the instantiators can be more aptly compared to transformation rules as they are first of all rule-based and second operate on the artefact model, but are themselves not part of it. The **common VIL expression language** represents a wide range of expressions from simple calculations over artefact operation and instantiator calls up to rather complex composite expressions. The expression language relies on the operations and operators provided by the VIL type system.

The two VIL languages are realized on top of the common expression language. Both languages follow a textual approach to the specification of artefact and product instantiation and support batch processing. Our definition of the syntax of the VIL languages draws upon typical concepts used in programming languages, in particular Java, build languages such as *make*, template languages such as *xtend* as well as expressions inspired by IVML and the Object Constraint Language (OCL) [6]. We adapt these concepts as needed to provide additional operations required in variability implementation, such as the integration with a variability model or an explicit artefact model.

We will use the following styles and elements throughout this section to illustrate the concepts of the IVML:

- The syntax as well as the examples will be illustrated in `Courier New`.
- **`Keywords`** will be highlighted using bold font.
- *`Elements and expressions`* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number.

- Statements will be separated using semicolon "`;`" (most other language concepts may optionally be ended by a semicolon).
- Different types of brackets will be used to indicate lists "`()`", sets "`{}`", etc. This is closely related to the Java programming language.
- We will indicate comments using "`//`" and "`/* ... */`" (cf. Java).

We will use the following structure to describe the different concepts:

- **Syntax:** this is the syntax of a concept. We will use this syntax to illustrate the valid definition of elements as well as their combination.
- **Description of syntax:** provides the description of the syntax and the associated semantics. We will describe each element, the semantics and their interaction with other elements in the model.
- **Example:** the concrete use of the abstract concepts is illustrated in a (simple) example.

In Section 3.1, we will describe the specific concepts of the VIL build language which is responsible for specifying the overall variability instantiation process of an entire (hierarchical) product line. In Section 3.2, we will describe the concepts of the VIL template language, which provides the means to describe the instantiation of a single (textual) artefact. Basic concepts of the VIL build and the VIL template language are rather similar (also to IVML) in order to simplify learning and application of these languages. In Section 3.3, we will detail the common expression language which is part of both, the VIL build and the VIL template language. In particular, we will detail the type system, i.e., the built-in types, their individual operations and the default instantiators that are part of the VIL implementation.

3.1 INDENICA Variability Build Language

In this section, we describe the concepts and language elements of the VIL build language in detail. This language aims at specifying the variability instantiation process of a whole (hierarchical) product line (as opposed to the instantiation of a specific artefact type covered by the VIL template language).

However, the VIL build language focuses on the implementation and instantiation of variabilities rather than on the entire build process of a whole system. Thus, the VIL build language is intended as an extension to existing build languages, i.e., it shall be integrated with those languages rather than replacing them.

3.1.1 Reserved Keywords

In the VIL build language, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by the keywords of the common VIL expression language given in Section 3.3.1.

- **`@advice`**
- **`const`**
- **`exclude`**
- **`extends`**
- **`execute`**

- **import**
- **instantiate**
- **join**
- **load**
- **properties**
- **protected**
- **requireVTL**
- **rule**
- **version**
- **vilScript**
- **with**

3.1.2 Scripts

In the VIL build language a script (**vilScript**) is the top-level element. This element is mandatory as it identifies the production strategies to be applied to derive an instantiated product. The definition of a script requires a name, as a basis for referring among VIL build scripts and a parameter list specifying the expected information from the execution environment such as the actual configuration or the projects to work on. In order to realize the necessary capabilities required for implementing the hierarchical product line capabilities of the EASy producer tool, at least the source project to instantiate from, the target project to be instantiated and the actual variability configuration must be passed to a VIL build script.

Basically, VIL may refer to all visible configuration settings in a variability configuration, more precisely to the actual values of frozen decision variables (and their underlying structure). In order to make this integration explicit, these decision variables may be directly referred in VIL by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. This may complicate the development of VIL scripts as actually unknown identifiers will at least lead to a warning. To support the domain engineer in specifying valid build scripts, VIL provides the **advice** annotation specifying the name for the IVML models used in the VIL build scripts. Qualified names resolvable via the **advice** annotation do not lead to warnings in the VIL editor. As an explicit version number may be stated for VIL scripts (akin to IMVL models), also advices and model imports may be version-constrained.

Optionally, a VIL script may extend another VIL script, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

Syntax:

```
//imports
@advice(ivmlName)
vilScript name (parameterList) extends name1 {
  //optional version specification
```

```
  //loading of variable values from an external source
  //variable definitions
  //rule declarations
}
```

Description of syntax: the definition of a build script consists of the following elements:

- First, all referenced scripts must be imported. We will detail the import syntax in Section 3.1.4.
- Optional advices declaring the underlying variability models.
- The keyword **`vilScript`** defines that the identifier *name* is defined as a new build script with contained production strategies.
- The parameter list denotes the arguments to be passed to a VIL script for execution. When executed by EASy, at least the source project(s), the target project, and the variability configuration must be passed in. Source and target project may be identical in case of (traditional) in-place instantiation. In case of multiple source (predecessor) projects, the source parameter shall be given as a collection. During processing, subsequent predecessors may be accessed via the `Project` type. However, further named parameters may be given upon an explicit invocation from an external call, e.g., an integration with a build language such as ANT or Maven. If given as first parameters in sequence project-configuration-project, the parameters will be bound independently of their name and, thus, the parameters can be named arbitrarily. Otherwise (due to the option of named parameters), the sequence of the parameters may be arbitrary but the parameters must exactly be named `source`, `config` and `target`.
- A VIL build script may optionally extend an existing (imported) VIL script. This is expressed by **`extends`** $name_1$, whereby $name_1$ denotes the name of the extending script.
- Production strategies are described within the curly brackets.

Example:

```
@Advice(YMS)
vilScript YMSBuild(Project source, Configuration config,
                   Project target){
  /* Go on with the production strategies for YMS here */
}
```

Please note that the types shown above such as `Project` or `Configuration` will be explained in detail in the next section. Further, a build script for multi-product

lines may require a container of projects (see Section 3.1.5.4), while single project parameter is sufficient for a traditional product line build.

A script defines the following two implicit variables:

- `OTHERPROJECTS` is a collection variable which contains the artefacts created by the execution of VIL build scripts in other projects.
- `SCRIPTDIR` is a String variable containing the location of this script in the file system.

3.1.3 Version

Akin to IMVL, VIL build specifications may optionally be tagged with an explicit version number in order to support product line evolution. Evolution of software may yield updates to projects, IVML models and build scripts so that scripts of different versions may exist and need to be clearly distinguished.

Syntax:

```
// Declaration of the version of a VIL build script.
version vNumber.Number;
```

Description of Syntax: A version statement consists of the following elements:

- The **`version`** keyword indicates a version declaration. At maximum one version declaration may be given in a VIL build file at the very first position within a VIL build script.
- *`vNumber.Number`* defines the actual version of the project (here only two parts prefixed by a “v”). At least one number must be given and no restriction holds on the amount of sub-version numbers.
- A version statement ends with a semicolon.

Example:

```
vilScript YMSBuild(Project source, Configuration config,
                   Project target){
    version v0.1.4;
    ...
}
```

3.1.4 Imports

The production strategies for a variability instantiation build process may be defined in a single VIL build script or may be reused from other (existing) build scripts. Therefore, VIL build scripts may be imported. In order to support also the evolution of product line build specifications, VIL allows the specification of version-restricted imports. Imports make the production strategies defined in the specified build file accessible to the importing build script.

Syntax:

```
// Unconstraint and constraint imports.
import name;
import name with (version op vNumber.Number);
```

Description of Syntax: An import[1] of a build scripts consists of the following elements:

- Importing a build script starts with the keyword **`import`**. Multiple imports may be given in a VIL build file directly at the beginning of the script file.
- *`name`* (given in terms of a VIL identifier) refers to the name of the build script to be imported. However, multiple scripts with identical names and versions may exist in a file system, in particular in hierarchical product lines. Thus, imports are determined according to the following **hierarchical import convention**, i.e., starting at the (file) location of the importing script (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). In addition, sibling folders of the folder containing the importing model and predecessor projects are considered[2]. Similar to Java class paths, additional script paths may be considered in addition to the immediate file hierarchy.
- An optional restriction of the import in terms of versions. This is indicated by the keyword **`with`** followed by a parenthesis containing the restrictions. A restriction is stated by the keyword **`version`**, a comparison operator (==, >, <, >=, <=) and a version number. An and-clause of multiple restrictions may be given in the parenthesis separated by commas.
- An import statement ends with a semicolon.

Example:

```
vilScript YMSBuild(Project source, Configuration config,
                   Project target){
    version v0.1.4;
    import generics with (version >= v1.12);
}
```

VTL templates may not be imported but restricted with respect to their version. The syntax looks similar to imports, but the syntax is adjusted to the actual way VTL

[1] Actually, this syntax differs from IVML due to technical reasons in xText.

[2] Actually, EasY-Producer stores the imported parent product line models in individual subfolders (starting with a "."), i.e. possibly sibling folders of a model.

templates are called from VIL. Using the syntax below, VTL scripts with name “name” are restricted to the given version akin to the import of VIL scripts.

```
restrictVTL "name" with (version op vNumber.Number);
```

3.1.5 Types

Basically, the VIL build language is a statically typed language with partially postponed type checking at runtime as we will detail below. Thus, the VIL build language provides a set of formal types to be used in variable declarations or parameter lists. We distinguish between basic types, configuration types, artefact types, and container types.

3.1.5.1 Basic Types

The basic types in the VIL build language correspond to the basic types of IVML, i.e., Boolean (`Boolean`), integer (`Integer`), real (`Real`) and string (`String`) with their usual meaning.

Boolean constants are given in terms of the keywords true and false. Integer constants are stated as usual numbers not containing a “.” or an exponential notation. Real constants must contain the floating-point separator “.” or may be given in exponential notation. Strings are either given in quotes or in apostrophs and may contain the usual escape sequences including those for line ends, quotes and apostrophs.[3]

3.1.5.2 Configuration Types

A configuration type denotes the representation of IVML configuration elements in VIL. However, due to the nature of VIL, we need only access to the configuration and the structure of an IVML model rather than to all modelling capabilities. Thus, VIL provides a specific set of built-in configuration types. The actual instance of a configuration is passed into a VIL build script in terms of a script parameter. Configuration types cannot be directly created in a VIL script and must not modify the underlying IVML model.

The entry point to a configuration in terms of an IVML model is the type `Configuration`. It provides access to all frozen decision variables and attributes. In particular, `Configuration` allows creating projections of a given configuration in order to simplify further processing. Further, it provides access to IVML type declarations such as compounds or enumerations and their value. This is represented by the `IvmlElement` and its subtypes. An `IvmlElement` represents IVML concepts in a generic way and provides access to its (qualified) name, its (qualified) type name and the configured value. Specific subtypes of `IvmlElement` are `DecisionVariable`, Attribute and `IvmlDeclaration`, each providing with more specific operations as we will discuss in detail in Section 3.4.4.

[3] Strings delimited by quotes may contain apostrophs, strings delimited by apostrophs may contain quotes.

3.1.5.3 Artefact Types

Artefact types represent the different categories of artefacts used in the artefact model. Some artefact types are built-in and part of the VIL implementation, while further types can be defined in terms of an extension of the artefact model. In this section, we will discuss only the predefined types. Please refer to the EASy developers guide on how to define more specific artefact types (as well as how to integrated instantiators implemented in a programming language).

The type `Project` is a mapping of a physical project (Eclipse) into VIL and provides related operations such as mapping paths between the source and the target project for instantiation.

A `Path` is a predefined type of the VIL artefact model although it is not an artefact by itself. A `Path` represents a relative file system path and may possibly contain wildcards. A path is specified in terms of a `String` in VIL and is automatically converted into a `Path` or an artefact instance depending on the actual use. In more detail, paths are specified according to the ANT [9] conventions, i.e., using the slash as path separator and wildcards for patterns. The following wildcards are supported: `?` for a single character (excluding the path separator), `*` for multiple characters (excluding the path separator) and `**` for (sub) path matches.

`Artifact`[4] is the most common artefact type and root of the VIL artefact hierarchy. The predefined `Artifacts` have also predefined methods. For example, they allow to delete the artefact (if possible at all), or to obtain access to its plain textual or binary representation. VIL provides a set of built-in artefact types such as `FileArtifact` and `FolderArtifact` which are both `FileSystemArtifacts`. Further, VIL provides more specific artefact types such as the `VtlFileArtifact` representing VIL template files (see Section 3.4.5) or the `XmlFileArtifact` representing parsed XML files with a substructure of specialized fragment artefacts such as `XmlElement` or `XmlAttribute`. Please note that artefact instances are assigned in a polymorphic way, i.e., while a `FileArtifact` may be specified as type in a VIL script, it may actually contain a more specific type. However, pattern paths, i.e., paths containing wildcards, will not be turned into artefact instances.

3.1.5.4 Container Types

VIL provides three container types, sequences (keyword `sequenceOf`), sets (keyword `setOf`) and associative containers (keyword `mapOf`). Container types are generic with respect to their content type(s) and, similarly to IVML, the content type must be stated explicitly, such as `sequenceOf(Integer)` or `setOf(DecisionVariable, FileArtifact)`.

Sequences may contain an arbitrary number of elements of a given element type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. In sequences, elements can be accessed by their position in the container using an index (`[`*`index`*`]`). In VIL, indexes start at zero and run until the number of elements in the container minus one (as in Java and many other

[4] We adopted US English in the implementation of VIL.

languages). Collections typically occur as results of operations, rule, or instantiator executions. In addition, they can be explicitly initialized using type-compatible expressions of the appropriate dimension as shown follows

```
sequenceOf(Integer) someNumbers = {1, 2, 3, 4, 5};
setOf(Integer, Integer) somePairs = {{1, 2}, {3, 4}};
```

A `Map` represents an associative container in VIL, i.e., a container which relates a keys to associated values. In particular, it allows retrieving the value assigned to a key via the `get` operation and the `[]`-Operator (`[key]`). Basically, associative containers are intended to simplify the translation of IVML-identifiers to implementation-specific identifiers in individual artefacts. Therefore, VIL associative containers can be explicitly initialized in terms of key-value-pairs using type-compatible expressions

```
mapOf(String, String) idTranslation
  = {{"nrOfProcessors", "procCnt"}, {"nrOfNodes", "nodeCnt"}};
```

VIL supports a set of operations specific for container types, e.g., excluding, projecting, or collecting elements in a container, etc. We will introduce the full set of operations in Section 3.4.3.

3.1.6 Variables

A variable provides name-based access to a value of a certain type (see Section 3.1.5), similar to variables in programming languages.

In VIL, the value of a variable can be modified at any time (in contrast to build languages such as ANT [9] where a value of a property can be set only once). In addition, a variable may be declared to be constant so that a value can be set only once and not be modified afterwards. Variables may be of global scope, i.e., directly defined within a VIL script or they may be local (within rules, see Section 3.1.6).

Syntax:

```
// Declaration of a variable.
Type variableName1;
Type variableName2 = value;
const Type constantName = value;
```

Description of Syntax: The declaration of variables consists of the following elements:

- The ***Type*** defines the type of the variable being declared.
- The identifiers *variableName*$_1$, *variableName*$_2$ and *constantName* are the names of the declared variable or constant, respectively.
- The optional keyword *const* indicates that a variable can be defined only once and the value must not be redefined.
- A variable may optionally be initialized by a value or an expression, which evaluates to a value of the given type.

- Variable declarations end by a semicolon.

Parameters of VIL build scripts are declared akin to variables, but without an initial value.

Example:

```
Integer numberOfCompilerProcesses = 4;
sequenceOf(Project) sources;
sequenceOf(Project, DecisionVariable) mapping;
```

Variables may be referred in Strings such as path patterns. A variable reference is stated as `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in Strings in the form `${expression}`. When applying the respective String, variable, and expression references are substituted with their actual value.

3.1.7 Externally Defined Values of Global Variables

Global variables or constants are defined as part of a VIL script. The value of a global variable or constant may be specified by an external source, e.g., to customize the build script according to the build environment (similar to properties in ANT [9]). For externally defined values of variables, initial values are not needed, in particular also not for constants. Externally specified values are subject to automated type conversion and variable reference or VIL expression substitution based on the VIL script arguments. Multiple external property files are processed in sequence so that the variable values defined by external files listed before are overwritten (accidental constant redefinition will lead to an execution error).

Syntax:

```
// Loading the values of global variables
load properties "path";
```

Description of Syntax: Loading values from an external file consists of the following elements:

- The keywords ***load properties*** indicates that the values of global variables shall be loaded from an external file. Multiple load statements may be given in a VIL script directly after the version statement.
- The path points to the file containing the initial values of the variables. Relative paths are interpreted relative to the target project of the VIL script. Also absolute paths may be given, in particular using variable references as described in Section 3.1.6. The file must be given in Java properties format, i.e., each line specifies the value of a specific variable in the following form

  ```
  variableName = value
  ```
- Loading values from an external file ends by a semicolon.

Example:

```
load properties "globalVariables.properties";
```

3.1.8 Rules

Build rules are used in VIL to specify individual production strategies, reusable build steps to be used within production strategies or, as the main entry point into the build process. Akin to *make* [8], VIL-rules may have preconditions, which must be fulfilled in order to enable the rule. However, VIL-rules may also explicitly define postconditions, which guard the result of the rule execution.

VIL rules may have **parameters** in order to parameterize the specified variability instantiation. These parameters must either be bound by the calling rule or, in case of the main entry rule, by the VIL build script itself.

Preconditions may be given in terms of path-patterns, an individual artefact, an artefact collection or rule calls. While an arbitrary number of rule calls may be given as precondition, at most one path pattern, artefact or artefact collection may be given as first precondition[5].

- A path pattern follows the (pattern) rules of ANT path specifications already described in Section 3.1.5.2. For example, "`$target/bin/**/*.class`" requires the existence of at least one Java bytecode file in the `bin` folder of `$target` (assuming that `$target` refers to the target project). Used as a rule precondition, a path pattern requires that the matching file artefacts exist and are up-to-date (akin to Make rule preconditions [8] but with extended pattern matching capabilities).
- An artefact (collection) is given in terms of a variable or a VIL expression evaluating to exactly one artefact (collection) instance. In a precondition, the denoted artefact(s) must exist and be up-to-date.
- A rule call (rule name with argument list) represents an explicit rule dependency and must be executed successfully in case that the preconditions of the stated rule are valid. The execution results of a rule call become available as an implicit variable in the rule body under the name of the called rule.

The optional **rule postcondition** is given in terms of a path pattern, an individual artefact or an artefact collection[5]. Postconditions are evaluated if the preconditions are met and the body of the rule is executed successfully. A rule completes successfully, if also the (optional) postcondition is met.

Rules may explicitly depend on each other in terms of the rule calls described above. Further, **implicit rule dependencies** are expressed via the first (non-rule call) pre- or postcondition (akin to *make* rules [8]). If a path matching precondition[5] for rule r_0 is not fulfilled, the VIL build language execution environment will aim at fulfilling the precondition by (recursively) searching for rules r_i with a postcondition indicating that the successful execution rule r_i contributes to the unmet precondition of rule r_0.

[5] Future work on VIL may relax this condition and even extend the current file path notation to more generic artifact model path expressions also involving fragment artefacts etc.

Ultimately, the possibly contributing rules r_i are executed (including their implicit rule dependencies) and the precondition of rule r_0 is checked again, and, on fulfilment, also r_0 is executed. If finally the precondition of r_0 is not fulfilled, r_0 is not considered for execution.

The rule body specifies the individual steps to be executed if the preconditions are met. A rule body may contain variable declarations, (assignment) expressions, explicit rule calls (not relevant as preconditions), instantiator calls, execution of system commands, or iterated execution of the previous elements. We will first describe the syntax of rules and describe then the individual statements available for specifying rule bodies.

If no path matching precondition is given, the rule body is executed once. If a path matching precondition is present, one or multiple artefacts may match that precondition and for each of these artefacts a corresponding output artefact may be required by the postcondition (if specified). However, the related match conditions may directly be used in the rule body but then possibly lead to a (superfluous) reinstantiation of the related target artefacts. In order to avoid reeinstantiation and to allow for optimizing the variability instantiation, the VIL build language offers two ways of executing the rule body. These two ways are:

1) Passing the right hand side matches as an implicit collection variable called `RHSMATCH` to the rule body. This is more appropriate for instantiations which may operate on multiple artifacts and consider dependencies among artefacts by themselves, such as a Java compiler.
2) Implicitly iterating over the matched pairs of left hand and right hand side artefacts. Then rule body is executed iteratively over all matching precondition artefacts. In order to address the actual artefact to be processed as well as its expected resulting artefact, the implicit variables `LHS` (in case of a matching precondition) and `RHS` (in case of a matching postcondition) will be made available to the loop body. This way is more appropriate for single artefact instantiations, such as calling a pre-processor or a C compiler.

All rules return implicitly their execution results consisting of two sequences,

- `result` containing the immediately modified artefacts by that particular rule.
- `allResults` containing the modified artefacts by all dependent rule calls.

Further, the artefacts modified by executed rules are be successively collected in the implicit global collection variable `OTHERPROJECTS`.

Syntax:

```
protected name (parameterList) = postcondition :
                preconditions {
// LHS/RHS/RHSMATCH may be available in the body
//variable declarations
//rule, instantiator, artifact or system calls
//iterated execution
}
```

Description of Syntax: A rule declaration consists of the following elements:

- The optional keyword **`protected`** prevents that this rule is visible from outside so that such rules cannot be used as an entry point (for example, in ANT [9] this is expressed by a target name starting with the minus character). This does not affect the internal accessibility of rules via imports and rule call.
- The *`name`* allows identifying the rule for explicit rule calls or for script extension.
- The *`parameterList`* specifies explicit parameters which may be used as arguments for precondition rule calls as well as within the rule body. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameter must either be bound by the calling rule or, in case of the main entry rule (usually called `main`), by the VIL build script itself (same parameter sequence and assignable values in both, the template and the main sub-template).
- The first three parts may be omitted in case of anonymous rules which are only executed due to implicit dependencies and not available to explicit rule calls.
- The optional postcondition specifies the expected outcome of the rule execution. A postcondition may be a path match, an artefact or an artefact collection. In case of a path match, the implicit variable RHS will be made available to the rule body.
- The optional preconditions specify whether the rule is considered for execution. The first precondition (made available as implicit variable LHS to the rule body) may be a path match, an artefact or an artefact collection. The following preconditions may be explicit rule calls. The execution results of the preconditions will be made available to the rule body in terms of implicit variables with names of the called rules and the rule return type described above.
- The rule body is specified within the following curly brackets.

Example:

```
produceGenericCopy(FileArtifact x, FileArtifact y) = y : x
{
    x.copy(y);
}

compileGoal() = "$target/bin/*.class" : "$source/*.java"
{
    javac(RHS, LHS);
}
```

The rule body specifies the individual steps to be performed in order to fulfil the rule postcondition (if stated). A rule body may contain variable declarations, (assignment) expressions, explicit rule calls, instantiator calls, execution of system commands or iterated execution of these elements. The statements (ended by a semicolon or a statement block) given in a rule body are executed in the given sequence. We will discuss these individual elements in the following subsections.

3.1.8.1 Variable Declarations

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within nested blocks. Basically, a variable declaration within a rule body follows the same syntax as global variable declarations discussed in Section 3.1.6.

3.1.8.2 Expressions

Expressions such as value calculations or execution of artefact operations may be used within a rule body as a guard expression or as a variable assignment. Please note that we will detail the VIL expression language in Section 3.3, as the expression language is common to both, the VIL build language and the VIL template language.

- Guard expressions constrain the execution of the remaining statements in a rule body, i.e., the expression must be evaluated successfully in order to continue the execution of the rule.
- In a variable assignment, the expression on the right hand side of the assignment operator "=" must be evaluated successfully in order to assign the evaluation result of the right side to the variable specified on the left side.

3.1.8.3 Calls

A call leads to the execution of another build language rule, an instantiator or an artefact operation. We will discuss three types of calls in this section, as they are represented by the same syntax. However, the most extreme call of a (blackbox) instantiator, namely the execution of an operating system command (including operating system scripts) follows a slightly different syntax. We will discuss operating system commands in Section 3.1.8.4.

The syntax of rule calls, instantiators or artefact operations looks as follows:

operationName(*argumentList*)

whereby arguments are expressions separated by commas. Calls may return values of different type.

Rule Calls

An explicit rule call is stated in terms of the name of the rule and the arguments matching the parameter list of the target rule. A rule call leads to the execution of a build language rule defined in the same script, one of the extended scripts or an imported script. As rules with the same signature consisting of name and parameter list are shadowed by the extension, rules in extended scripts may explicitly be called by

```
super.operationName(argumentList)
```

Operations defined in imported scripts may be denoted by their qualified name as `operationName`, i.e., prpending the import path until the defining model.

Instantiator Calls

Basically, the VIL build language aims at defining the production flow for instantiating generic artefacts for a software product line. In contrast, the VIL template language aims at specifying the individual actions to instantiate an individual (generic) artefact. Further instantiators may be given in terms of (wrapping) Java classes in order to make programming language compilers, linkers, or legacy instantiators available. Such instantiators may provide information about their execution, in particular the created artefacts.

In the VIL build language, all these types of instantiators are mapped transparently to one kind of statement, the instantiator call:

```
operationName(argumentList)
```

Basically, an instantiator call looks similar to a rule call, i.e., a name with a parameter list, but it (typically) returns a collection of artefacts (or even nothing in case of wrapped blackbox instantiators). Instantiators may be rather generic (such as the built-in instantiator for the VIL template language) and may offer to pass an arbitrary number of arguments (e.g., those defined by a VIL template. Therefore, depending on the instantiator, named arguments (`parameterName = valueExpression`) may pass arbitrary VIL instances to an instantiator in a generic way.

We will detail the built-in instantiators in Section 3.4.5.12. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize an instantiator.

Artefact Operation Calls

Artefact operations provide information on an individual artefact, its fragments or even enable the manipulation of artefacts. Basically, an artefact operation is executed on a variable or expression, which evaluates to an artefact type. An artefact operation can be expressed (akin to IVML and OCL) in two different ways, using the artefact as first argument

```
operationName(artefact, argumentList)
```

or in object-oriented style

```
artefact.operationName(argumentList)
```

Basically, a String can be automatically converted into a Path or an `Artifact`. Similarly, a Path can be transparently converted into an `Artifact`. However, in some cases, also an explicit creation of an artefact of a certain type may be required. Typically, the individual artefact types support the following constructors

```
new ArtefactType(String)
```

for obtaining a specific artefact specified by its path. Please note that artefacts are associated with creation rules detailed in Section 3.4.5. Basically, file artefacts (regardless of whether they physically exist or only the path is known) are polymorphically determined according to their file name extension, e.g., a file with extension `xml` is considered to be a `XmlFileArtifact`. However, pattern paths, i.e., paths containing wildcards, will not be turned into artefact instances. Further, content-specific rules may apply depending on the specific artefact type. If no such rule applies, a basic `FileArtifact` is created as the default fallback. Thus, the underlying mechanisms of the VIL artefact model will check whether the creation of that instance (regardless of whether the underlying file exists or not) is actually possible or not. If the creation fails, also the containing rule will fail. Folders are created transparently, e.g., when the underlying file beyond a file artefact is created. The constructor

```
new ArtefactType()
```

allows to obtain a temporary file or folder artefact. Unless not renamed, this artefact will be automatically deleted after terminating the execution of the VIL script.

The modifications to a VIL artefact instance will automatically be synchronized with the underlying artefact upon the end of the lifetime of the related variable, e.g., when the execution of the containing scope of a local variable ends.

We will detail the built-in artefact operations in Section 3.4.5. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize own artefact types and related operations.

Operation Resolution

While determining the applicable rules, instantiators, or artefact operations, the VIL type system considers in the following sequence

(1) Exact match of argument types and parameter types.
(2) Assignment compatible argument types and parameters.
(3) Automatic conversions specified as part of the implementation of VIL types and artefacts, e.g., the implicit conversion of a `String` to a `Path` or a `String` to an `Artifact`. If multiple conversion operations may support the required conversion, the most specific one with respect to the type hierarchy of the source type will be applied. Details on the type system and the available conversions will be discussed in Section 3.3.

The operation types discussed in this section will be resolved according to the sequence below:

(1) Rule calls

(2) Instantiator calls
(3) Artefact, configuration type, and basic type operations

Further, the VIL runtime environment performs dynamic dispatch, i.e., the operation determined and bound at script parsing time will be reconsidered with respect to the actual types of parameters and the best matching operation will dynamically be determined (similar to dynamic dispatch in Xtend [2]). This avoids the need for explicit type checking or large alternative decision blocks.

3.1.8.4 Operating System Commands

The VIL build language is also able to execute the most basic form of a blackbox instantiator, namely operating systems calls or scripts. However, the syntax for system calls differs slightly from the other call types discussed in Section 3.1.8.3 as operating system commands may require explicit path specifications.

`execute` *`identifier`*`(`*`argumentList`*`)`

whereby *`identifier`* must denote a variable which evaluates to a `String` or a `Path`. This enables that operating system calls can be composed at script execution time or determined using external values (see Section 3.1.7). However, the related command or script is executed, but the created artefacts are not tracked by the VIL execution environment.

3.1.8.5 Iterated Execution

Finally, all statements available in a rule body may explicitly be executed in iterative fashion, e.g., to apply a sequence of instantiator calls explicitly to a container of artefacts. Therefore, the VIL build language offers a dedicated loop statement. However, this expression called `map` in the VIL build language is different from typical programming language loops as it collects the result of its execution in terms of modified artefacts (similar to a rule). The results produced by a map expression are determined by the last standalone expression in a map body. The evaluation results of these expressions are collected in a sequence of type of the expression. The results can be assigned to a variable. If there is no such expression, map will behave like a typical for-loop. However, due to its character as an expression, a map must be used within an expression and terminated by a semicolon.

Syntax:

```
map(names = expression) {
//variable declarations
//rule, instantiator, artefact or system calls
//iterated execution
};
```

Description of Syntax: An iterated execution consists of the following elements:

- The keyword **`map`** followed by parenthesis defining the iterator variables.
- The *`names`* denoting the names of the variables used by the map expression iterate. The number, type and the contents of the iterator

variables are implicitly defined by the related `expression`. Typically, the expression will lead to a container with one parameter type so that map will iterate over that collection using exactly one variable of the element type of the container. However, as we will discuss in Section 3.1.8.6, the `join` expression may return a multi-dimensional container, which then needs multiple iterator variables.

- The **`map`** consists of a block determining the statements to be executed in for an individual iteration. The *`names`* denoting the iterator variables shall be used within the block.

Example:

```
map(d : config.variables()) {
    // operate on the iterator variable d of type
    // DecisionVariable (see Section 3.4.4.6)
};
```

3.1.8.6 Join Expression

One specific expression in the VIL build language is particularly intended to be used with the `map` iteration statement, namely the `join` operation. However, as `join` is an expression, it may be used as an usual expression, e.g., on the right hand side of a value assignment to a variable.

This operation is inspired by database joins, e.g., as usual in SQL. The `join` operation allows combining containers of different VIL types, in particular elements from the variability configuration with source or target artefacts. Depending on type of the specified expression types, the `join` operation returns a typed sequence containing the results.

Syntax:

`join(`*`name`*$_1$*`:expression`*$_1$**`, `***`name`*$_2$*`:expression`*$_2$**`) with (`***`expression`***`)`**

Description of Syntax: A VIL join expression consists of the following elements:

- The keyword **`join`** followed by one parenthesis defines the containers to be joined and the related iterator variables (*`name`*$_1$*`, name`*$_2$).
- *`name`*$_1$*`, name`*$_2$ denote variables used by the join expression iterate over the containers given in *`expression`*$_1$ and *`expression`*$_2$. Without further restriction, the result will be a collection of pairs on the types parameterized by the types of *`expression`*$_1$ and *`expression`*$_2$*.* The keyword `exclude` used before one of the names leads to a left- or right-sided join, thus restricting number of parameters of the resulting collection to one.
- The third `expression` specifies the join condition, i.e., an expression involving *`name`*$_1$ and *`name`*$_2$ to select the relevant results from the cross

product of *name*$_1$ and *name*$_2$, and to effectively reduce the size of the result.

Example:

```
// work on those decisions and artefacts where a certain
// string composed from the decision name occurs in the
// artefact (and may be substituted by an instantiator)
map(d, a :
  join(d:config.variables(), a:"$source/src/**/*.java")
  with (a.text().matches("${" + d.name() + "}")) {
    // operate on decision variable d and
    // related artifact a
}
```

3.1.8.7 Instantiate Expression [6]

In addition to the instantiation of individual projects, VIL is specifically intended to support the instantiation of hierarchical and multi product lines. Therefore, it is necessary to execute scripts in other projects, if required on projections of the configuration and with specific target project. Basically, this is possible by explicitly calling the main rule of the predecessor projects statically, i.e., to reference them by their qualified name. However, it is not possible to iterate (recursively) over the predecessor projects and to instantiate them. Further, the references to the predecessor projects would be static and not subject to possible variability. Thus, we introduce the instantiate expression, which allows executing a VIL rule in a project allowing to dynamically referring to VIL scripts in (other) projects. Further, as a side effect, the instantiate expression also allows calling rules via dynamically composed qualified rule names. Please note, that due to the dynamic character, rule calls via the instantiate expression imply a higher overhead.

Syntax:

```
instantiate name (argumentList)

instantiate name (argumentList) with (version op
vNumber.Number)

instantiate name rule "ruleName"(argumentList) with
(version op vNumber.Number)

instantiate"ruleName" (argumentList)
```

Description of Syntax: A VIL instantiate expression consists of the following elements:

- The keyword **instantiate** followed by either the name of a variable containing the project to instantiate (form 1-3) or a string containing the qualified name of the rule to be executed (form 4).

[6] Please note that the instantiate expression is currently not completely implemented. In particular, EASy producer does currently not pass the predecessor locations to VIL. This will be changed in future version.

- The argument list of the rule to be executed in parenthesis. Please note, that in particular also the source project (in form 1-3 the project the script is defined for), the (relevant part of the) configuration and the target project (typically the calling project) must be given (following the VIL parameter conventions). Further, optional named arguments may be given.
- Finally, the version specification of the model to be instantiated may be given.

Example:

```
// instantiate the predecessor projects into target
vilScript a (Project source, Configuration config,
  Project target) {
  map(Project p: source.predecessors()) {
    instantiate p (p, config, target);
  }
}
```

3.2 VIL Template Language

In this section, we describe the concepts and language elements of the VIL template language in detail. In contrast to the VIL build language, which aims at specifying the instantiation of all artefacts of a product line, the VIL template language aims at specifying the instantiation of a single artefact.

3.2.1 Reserved Keywords

In the VIL template language, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by those of the common VIL expression language given in Section 3.3.1.

- **`@advice`**
- **`@indent`**
- **`const`**
- **`def`**
- **`default`**
- **`else`**
- **`extends`**
- **`extension`**
- **`for`**
- **`if`**
- **`import`**
- **`print`**
- **`switch`**
- **`template`**
- **`version`**
- **`with`**

3.2.2 Template

The template (**`template`**) is the top-level structure in the VIL template language. This element is mandatory as it defines the frame for specifying how to instantiate a certain artefact. Please note that exactly one template must be given in a VIL template file.

The definition of a template requires a name, which acts for referring among VIL templates and a parameter list specifying the expected information from the calling VIL build script such as the actual configuration and the target artefact (fragment). Please note that these two arguments must be provided to all VIL template scripts.

Basically, VIL may refer to all visible configuration settings in a variability configuration, more precisely to those actual values of decision variables (and their underlying structure), which are frozen. In order to make this integration explicit, these decision variables may be directly referenced in the VIL template language by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. To support the domain engineer in

specifying valid templates, also the VIL template language provides the **advice** annotation (see also Section 3.1.2).

Optionally, a VIL template may extend another VIL template, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

The VIL template language particularly aims at supporting generative and manipulative instantiation of generic artefacts. Therefore, the VIL template language provides capabilities for easily specifying and generating contents. However, as usual in software development, also VIL templates shall be formatted properly. In order to distinguish between intended formatting and whitespaces that shall not occur in the target artefact (fragment), the VIL template language is able to take the actual indentation into account (as specified in the **indent** annotation). Taking the formatting of the templates into account avoids postprocessing of the results, e.g., by formatting mechanisms [2]. We will discuss the indentation processing of the VIL template language as part of the content statements in Section 3.2.8.6.

Akin to programming languages, VIL templates may contain (global) variable declarations as well as sub-templates (similar to methods in object-oriented programming languages or functions in the structured programming paradigm).

Syntax:

```
//imports
//functional extensions
@advice(ivmlProjectName)
@indent(indentationSpec)
template name (parameterList) extends name1 {
  //optional version specification
  //variable definitions
  //sub-template declarations
}
```

Description of syntax: The definition of a VIL template consists of the following elements:

- Optionally, imported templates or functional extensions by Java classes are listed first.
- Optional advices declaring the underlying variability models. This annotation is similar to VIL (see Section 3.1.2).
- An optional indentation annotation enabling the VIL template execution to take the actual indentation into account when processing content statements. We will detail the use of the indentation annotation in Section 3.2.8.6 along with the content statement, which actually considers indentation information.
- The keyword **template** defines that the following identifier *name* defines a new artefact instantiation template.

- The parameter list denotes the arguments a VIL template requires when being executed. Basically, a VIL-template receives the underlying variability configuration and the target artefact as parameters. If given as first parameters in this sequence, the parameters will be bound independently of their name and, thus, the parameters can be named arbitrarily. Otherwise (due to the option of additional named parameters as mentioned below), the sequence of the parameters may be arbitrary but the parameters must exactly be named “config” and “source”. The arguments are subject to dynamic dispatch, i.e., either the most generic type `Artifact` may be used for the target artefact or a more specific type can be used. In the latter case, the instantiator statement in the VIL build script must also pass in a type-compliant artefact instance. Additional parameters may be defined which then must be stated in the calling VIL build script as named arguments.
- A VIL template may optionally extend an existing (imported) VIL template. This is expressed by **`extends`** *name*$_1$, whereby *name1* denotes the name of the extending script.
- The optional version specification, variable declarations and sub-templates are then stated within the curly brackets.

Example:

```
@advice(YMS)
template DbCreator (Configuration config,
  Artifact target){
  /* Go on with description of the artefact instantiation
  starting with a main sub-template and possibly further
  (imported) sub-templates */
}
```

3.2.3 Version

Akin to IMVL and the VIL build language, also the VIL template language can be tagged with an explicit version number in order to support evolution. The syntax for the version declaration is identical to the VIL build language as discussed in Section 3.1.3.

3.2.4 Imports

The description of the instantiation of a certain artefact type may be defined in a single VIL template (possibly including sub-templates) or may be composed from reusable sub-templates specified in other (existing) build scripts. Therefore, VIL templates may be imported. In order to support also the evolution of product line build specifications, also the VIL template language allows the specification of version-restricted imports. Imports make the sub-templates defined in the specified build file accessible to the importing template. The syntax of imports in VIL

templates is identical to imports in the VIL build language as discussed in Section 3.1.4.

3.2.5 Functional Extension

Sometimes, it is necessary to realize specific supporting functions such as calculations in terms of a programming language rather than in the template language itself. Therefore, similar to Xtend [2], the VIL template language enables external functions in terms of static Java methods, to call these methods from the template language and to use the results in VIL templates. Basically, the realizing classes are declared in VIL as extension and containing static methods are made available as they would be VIL operations[8]. However, methods with already known signatures will not be redefined.

Syntax:

```
extension name;
```

Description of Syntax: A functional extension in the VIL template language consists of the following elements:

- The keyword **`extension`** followed by a qualified Java *`name`* denoting the class to be considered. The referred class must be available to VIL through class loading. Contained static methods will be considered as extension methods for the VIL template language. Please note that the implementing method shall use only primitive Java types or (the implementation classes of the) VIL types discussed in Section 3.3.
- An extension declaration ends with a semicolon.

Example:

```
extension java.lang.System;
```

3.2.6 Types

Basically, the VIL template language is a statically typed language with some convenience in terms of postponed type checking at runtime akin to the VIL build language. Thus, the VIL template language provides a set of formal types available for variable declarations or parameter lists. VIL template language and VIL build language rely on the same type system and, thus, the VIL template language provides the same types as discussed in Section 3.1.5.

3.2.7 Variables

A variable provides named access to a value of a certain type similar to variables in programming languages. The semantic of variables as well as the syntax for declaring

[8] Additional classes may require special treatment in terms of class loading, in particular in OSGi and Eclipse due to specific class loaders per bundle. In such environments, the class loader being responsible for the extension classes must be explicitly registered with the VIL runtime environment (see `de.uni_hildesheim.sse.easy_producer.instantiator.model.templateModel.ExtensionClassLoaders`).

and using them in the VIL template language is identical to the VIL build language as discussed in Section 3.1.6 (except for the capability of defining variable values in an external file which is not available in the VIL template language).

Similar to the VIL build language, variables may be referred in Strings such as paths or content statements. A variable reference looks like `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in the form `${expression}`. When applying the respective element, variable and expression references are substituted by their actual value.

3.2.8 Sub-Templates (defs)

The actual instantiation of an artefact is given in terms of sub-templates (called `def` in the concrete syntax), i.e., named functional units with parameters and return types. One specific sub-template (usually called `main`) acts as the entry point into artefact instantation. Akin to the VIL build language, it receives the parameters of the containing template as arguments (in the same sequence).

The body of a sub-template specifies the individual steps to be executed for realizing the instantiation. Such a body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). We will first describe the syntax of templates and discuss then the statements available in sub-template bodies.

Syntax:

```
def name (parameterList) {
//variable declarations
//alternative, switch-case, loop
//content statements
}

def Type name (parameterList) {
//variable declarations
//alternative, switch-case, loop
//content statements
}
```

Description of Syntax: A sub-template declaration consists of the following elements:

- The keyword **`def`** indicates the definition of a sub-template.
- By default, the VIL template language aims at inferring the return type of a sub-template from the rule body. In the extreme case, individual statements may produce a rather generic value of type `AnyType`, which enables the use of the value without type checking at template parsing time and type checking at runtime. However, in some situations the template developer may explicitly want to do strict type checking at

template parsing time. This is enabled by specifying the optional return type for a sub-template.

- The *name* allows identifying the sub-template for calls, template extension or as `main` entry point.
- The *parameterList* specifies the parameters of a sub-template in order to parameterize the instantiation operations subsumed by the respective sub-template. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameters must either be bound by the calling sub-template or, in case of the main entry rule, by the VIL template itself (via identical names and assignable types, the template and the main sub-template).
- The rule body is specified within the following curly brackets.

Example:

```
def main(Configuration config, Artifact target) {
    // define artifact instantiation
}

def String valueMapping (DecisionVariable var) {
    // map the value of var to a String
    // explicit type checking is enforced
}
```

The sub-template body specifies the individual steps needed to instantiate an artefact. Such a rule body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). The statements (ended by a semicolon or a statement block) given in a sub-template body are executed in the given sequence. We will discuss these individual statement types in the following subsections.

The last statement executed in a sub-template body implicitly determines the return value of a sub-template.

3.2.8.1 Variable Declaration

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within statement-blocks such as loops or alternatives. Basically, a variable declaration within a sub-template body follows the same syntax as global variable declarations discussed in Section 3.2.7.

3.2.8.2 Expression Statement

Expressions such as value calculations or execution of artefact operations may be used within a sub-template body as a guard expression or as a variable assignment. Please note that we will detail the VIL expression language in Section 3.3, as the expression language is common to both, the VIL build language and the VIL template language. Thus, guard expressions and variable assignments as discussed in Section

3.1.8.2 are similarly available in the VIL template language. Further, similar call types as well as their (resolution) semantic as discussed in Section 3.1.8.3 are available in the VIL template language (of course, rule calls are replaced by template calls and operating system calls by calls to functional extensions). Template calls may be recursive. In the VIL template language, the resolution sequence is

1) Template calls
2) Artefact, configuration type and basic type operations
3) Functional extension calls

3.2.8.3 Alternative

In the VIL template language, alternatives allow choosing among different ways of instantiating an artifact, i.e., upon evaluating a condition the appropriate alternative to execute is determined.

The (return) type of an alternative is either the common type of all alternatives or `AnyType`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
if (expression) ifStatement
if (expression) ifStatement else thenStatement
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`if`** indicates the beginning of an alternative statement.
- The *`expression`* given in parenthesis determines whether the if-part (condition is evaluated to true) or the else-part (condition is evaluated to false) is executed. In particular, if the *`expression`* cannot be evaluated, evaluation will terminate the evaluation of the sub-template block before evaluating the alternatives.
- The *`ifStatement`* (or statement block enclosed in curly braces) is being executed when the *`expression`* is evaluated to true. A single statement must be terminated by a semicolon.
- The **`else`** part is optional, i.e., if else is used in an alternative, a following *`thenStatement`* is required. As usual, a dangling else is bound to the innermost alternative.
- The *`thenStatement`* (or statement block enclosed in curly braces) is executed if the *`expression`* is evaluated to false. Again, a single statement must be terminated by a semicolon.

Example:

```
if (config.variables().size() > 0) {
    // work on config
} else {
    // produce an empty artefact
}
```

3.2.8.4 Switch

The switch statement in the VIL template language is for (dynamically) mapping configuration elements to artefact elements rather than for influencing the control flow (as it is the case for the alternative statement). However, in case of larger mappings with (more or less) static content, we suggest using a map variable (see Section 3.1.5.4).

The (return) type of an alternative is either the common type of all cases or `AnyType`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
switch (expression) {
  expression1 : expression2,
  expression3 : expression4
}

switch (expression) {
  expression1 : expression2,
  default : expression5
}

switch (expression) {
  expression1 : expression2,
  expression3 : expression4,
  default : expression5
}
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`switch`** indicates the beginning of a switch statement. It is followed by an *`expression`* to switch over and the individual cases in a block of curly brackets.
- A case consists of an expression (*`expression`*$_1$ or *`expression`*$_3$ above) to be matched against expression. If an individual match succeeds, the related value expression will be evaluated and determines the (result) value of the switch statement. The implicit variable `VALUE` may be used within the value expression in order to refer to the evaluated value of *`expression`*.
- Optionally, a **`default`** case can be given which is considered if none of the previous cases matches. In that case, the expression stated behind the default will be evaluated and determines the (result) value of the switch statement.

Example:

```
switch (var.name()) {
    ”forkNumber” : VALUE + var.intValue() – 1,
    ”cpuNumber” : var.stringValue()
    //go on with further cases and a default value
    //if required
}
```

3.2.8.5 Loop

The for-statement in VIL enables the defined repetition of statements. Basically, it is rather similar to an iterator-loop in Java.

Syntax:

for (Type *var* **:** *expression***)** *statement*

for (Type *var* **:** *expression*, *expression*$_1$**)** *statement*

for (Type *var* **:** *expression*, *expression*$_1$, *expression*$_2$**)** *statement*

Description of Syntax: A loop statement consists of the following elements:

- The keyword **for** indicates the beginning of a for-loop statement. It is followed by a parenthesis defining the loop iterator, i.e., a variable to which successively all values of the *expression* are assigned. Therefore, *expression* must either evaluate to a set or a sequence.
- The statement (or statement block enclosed in curly braces) is then executed for each element in the collection specified by *expression* while the iterator *var* is successively assigned to each individual value in the collection.
- An optional separator expression *expression*$_1$ which is emitted (without line end) at the end of each iteration if further iterations will happen. Such a separator expression simplifies generating value lists or similar target artefact concepts.
- A second optional separator expression *expression*$_2$ which may only be stated if *expression*$_1$ is given. *expression*$_2$ is emitted (without line end) at the end of the last iteration. Such a separator expression simplifies generating value lists or similar target artefact concepts.

Example:

```
for (DecisionVariable var: config.variables()) {
    //operate on var
}

for (DecisionVariable var: config.variables(), ”, ”) {
    //print var (the separator will be added if needed)
}
```

```
for (DecisionVariable var: config.variables(), ",", ";") {
    //print var (the separator will be added if needed,
    //the second separator will be printed after the final
    //iteration)
}
```

3.2.8.6 Content

The content statement is used to generate the content of the target artefact. Basically, all characters given in a String (enclosed in a pair of apostrophs or quotes including appropriate Java escapes and line breaks) are emitted as output to the result artefact. Content statements executed in the course of template evaluation according to the control flow make up the entire content of the target artefact (fragment). A content statement may consist of multiple lines as part of the content. Thereby, variable references or IVML expressions are substituted as described for variables in Section 3.2.7.

Without further consideration, also the indentation whitespaces for pretty-printing a VIL-template will be taken over into the resulting artefact. In order to provide more control about the formatting, the annotation **`@indent`** allows specifying the number of whitespaces used as one indentation step (value `indentation`), the number of whitespaces to be considered in tabulator emulation (value `tabEmulation`) and also how many additional whitespaces (value `additional`, default is `1`) are used to indent the content statement, i.e., whether the following lines after the lead in character are further subject to indentation or not. Further whitespaces in the content are considered as formatting of the content itself and are taken over into the artefact. In addition, an optional numerical value can be specified at each content statement in order to programmatically indent the configuration by the given number of whitespaces.

Syntax:

```
"text"

print "text"

"text" | expression;

'text' | expression;
```

Description of Syntax: A content statement consists of the following elements:

- The optional keyword **`print`**, which indicates that no line end shall be emitted at the end of the content. If absent, implicitly a print-line is performed. The print form of the content statement is helpful in combination with the separator expression in loops.
- The lead in / lead out marker (apostrophe or quote) indicates the content statement and marks the actual content. Two forms are used to enable the use of the opposite character in content. A content statement may cover multiple lines of content.

- An optional indentation expression can be indicated by the pipe symbol and followed by a numerical *expression*. The numerical *expression* determines the amount of whitespaces to be used as mandatory indentation prefix for each individual line of the actual content statement. Only if the indentation expression (may be a constant or a true expression) is specified, a semicolon is required.

Example:

```
'CREATE DATABASE ${var.name()}'
'CREATE TABLE data' | 4;
```

3.3 VIL Expression Language

In this section, we will define the syntax and the semantics of the VIL expression language, which is common to the VIL build language and the VIL template language. Similar to IVML, expressions in the VIL languages are inspired by OCL. Thus, most of the content in this section is taken from OCL [6] or the IVML language specification [3, 7] and adjusted to the need, the notational conventions, and the semantics of the VIL languages, in particular also regarding the syntax of the iterators and the absence of quantors in VIL.

3.3.1 Reserved Keywords

Keywords in the VIL expression language are reserved words. That means that the keywords must not occur anywhere in an expression as the name of a rule, a template or a variable. The list of keywords is shown below:

- **and**
- **false**
- **new**
- **not**
- **or**
- **sequenceOf**
- **setOf**
- **super**
- **true**
- **xor**

In order to increase reuse among the VIL languages, the VIL expression language also provides the definition of common language concepts such as variable declarations and parameter lists. The related keywords were already listed in Section 3.1.1 and 3.2.1, respectively.

3.3.2 Prefix operators

The VIL expression language defines two prefix operators, the unary

- Boolean negation '**not**' and its alias '!'.
- Numerical negation '−' which changes the sign of a Real or an Integer.

Operators may be applied to constants, (qualified) variables or return values of calls.

3.3.3 Infix operators

Similar to OCL, in VIL the use of infix operators is allowed. The operators '+,' '−,' '*.' '/,' '<,' '>,' '<>' '<=' '>=' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the "+" operation on *a* (the *operand*) with *b* as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators '<,' '>,' '<=,' '>=,' '<>,' '**and**,' '**or**,' '**xor**', the return type is Boolean.

Operators may be applied to constants, (qualified) variables or return values of calls.

Please note that, while using infix operators, in VIL Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real. We will detail the operations on basic types in Section 3.4.2.

3.3.4 Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot operations: '.' (for operation calls in object-oriented style)
- unary '**not**', !(alias for **not**) and unary minus '–'
- '*' and '/'
- '+' and binary '–'
- '<', '>', '<=', '>='
- '==' (equality), '<>', '!=' (alias for '<>')
- '**and**', '**or**' and '**xor**'

'(' and ')' can be used to change precedence.

3.3.5 Datatypes

All artefacts defined by the extensible VIL artefact model as well as the various built-in types are available to the expression language and may be used in expressions. IVML elements are mapped into VIL via IVML qualified names. Figure 1 illustrates the VIL type hierarchy (not detailing the IVML integration through the configuration types). Below, we discuss the use of datatypes.

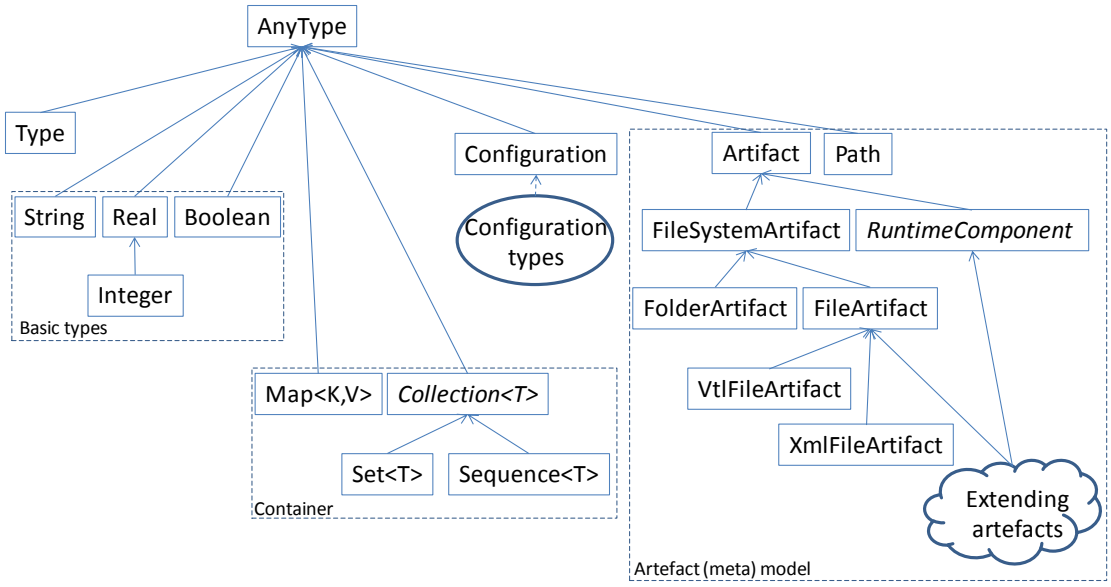

Figure 1: Overview of the VIL type system

3.3.6 Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- `AnyType` is the common superclass of all types. All types comply with `AnyType`. However, `AnyType` is not directly available to the user and used internally to denote expressions of unknown type to be resolved and checked while execution time of the specifying script.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies or if no type parameters are given (deferred to script evaluation time).

3.3.7 Side effects

In contrast to OCL, some constraint expressions in IVML may lead to side effects, in particular to modifications of artefacts and artefact fragments.

3.3.8 Undefined values

Basically, variables and IVML qualified names, i.e., links into a variability model, may be undefined. During evaluation of expressions at script or template execution time, undefined (sub-)expressions are ignored and do not lead to failing of rules or sub-templates.

3.3.9 Collection operations

The VIL artefact model and the IVML integration into VIL in terms of the `Configuration` type define many operations returning collections. Operating with collections is specifically meant to enable a flexible and powerful way of accessing information and for deriving artefacts.

Practically, collections in VIL are sets, sequences, or maps as introduced in Section 3.1.5.4. Collections are a basis for iterations in the VIL template language (Section 3.2.8.5) as well as for joins and map iterations in the VIL build language (Section 3.1.8.5). Therefore, we defined a set of basic operations on the artefact types and the `Configuration` providing convenient access through selection and filtering. However, with an increasing amount of information provided by the accessible types, more and more collection access operations and related changes to the VIL types will be needed. However, we refrained from a specific syntax for these operations as in IVML and rely on implicit iterator variables similar to other implicit variables in the VIL languages. One example is the **select** operation, which returns all elements of a collection complying to a certain Boolean selection expression. An example for an expression with a generic iterator is

```
configuration.variables().select(ITER.name().length() = 10)
```

Which returns those decision variables with a name of length 10 (`ITER` is the generic iterator variable implicitly provided by the VIL language for such specific collection operations)[9].

3.4 Built-in operations

Similar to OCL and IVML, all operations in VIL are defined on individual types and can be accessed using the "." operator, such as `set.size()`. However, this is also true for the equality, relational, and mathematical operators but they are typically given in alternative infix notation, i.e., 1 + 1 instead of 1.+(1) as stated in Section 3.3.3. Further, the unary negation is typically stated as prefix operator. Due to the VIL artefact model, the integration with IVML and the specific purpose of variability instantiation, the VIL types define a different set of operations than OCL/IVML[10].

In this section, we denote the actual type on which an individual operation is defined as the *operand* of the operation (called *self* in OCL). The parameters of an operation are given in parenthesis. Further, we use in this section the Type-first notation to describe the signatures of the operation.

Please note that those operations starting with "get" (Java-getters) are also available with their short name without "get" in order to simplify script and template creation,

[9] We are aware of the fact that VIL currently does not support using two iterators in the same expression. We will target this in a future revision of the language.

[10] An extended set of operations will be defined by future extensions of VIL.

e.g., the `getName()` operation is also available as its aliased operation `name()`. We will make this explicit by listing both operation signatures.

3.4.1 Internal Types

3.4.1.1 AnyType

`AnyType` is the most common type in the VIL type system. All types in VIL are type compliant to `AnyType`. In particular, `AnyType` is used as type if the actual type is unknown at parsing time and shall be determined dynamically at runtime. Therefore, `AnyType` can be assigned to variables of any type (but no specific operations can be executed on `AnyType`). `AnyType` can be converted automatically into a `String`.

3.4.1.2 Type

Type represents type expressions themselves and enables the type-generic selection type-compliant elements from collections.

3.4.2 Basic Types

In this section, we detail the operations for the basic VIL types.

3.4.2.1 Real

The basic type `Real` represents the mathematical concept of real numbers following the Java range restrictions for double values. Note that `Integer` can automatically be converted to `Real`.

- **Real + (Real r)**
  The value of the addition of *self* and the *operand*.
- **Real - (Real r)**
  The value of the subtraction of *r* from the *operand*.
- **Real * (Real r)**
  The value of the multiplication of the *operand* and *r*.
- **Real - ()**
  The negative value of the *operand*.
- **Real / (Real r)**
  The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.
- **Boolean < (Real r)**
  True if the *operand* is less than *r*.
- **Boolean > (Real r)**
  True if the *operand* is greater than *r*.
- **Boolean <= (Real r)**
  True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
  True if the *operand* is the same as *r*.
- **Boolean != (Real r)**
  True if the *operand* is not the same as *r*.
- **Boolean <> (Real r)**
- True if the *operand* is not the same as *r*.

- **Boolean == (Real r)**
  True if the *operand* is the same as *r*.
- **Real abs()**
  The absolute value of the *operand*.
- **Integer floor ()**
  The largest integer that is less than or equal to the *operand*.
- **Integer ceil()**
  The closest integer value that is greater or equal to the *operand*.
- **Integer round()**
  The integer that is closest to *the operand*. When there are two such integers, the largest one.

3.4.2.2 Integer

The standard type `Integer` represents the mathematical concept of integer numbers following the Java range restrictions for integer values. Note that `Integer` is a subclass of `Real`.

- **Integer + (Integer i)**
  The value of the addition of the *operand* and *i*.
- **Integer - (Integer i)**
  The value of the subtraction of *i* from the *operand*.
- **Integer * (Integer i)**
  The value of the multiplication of the *operand* and *i*.
- **Real / (Integer i)**
  The value of the *operand* divided by *i*. Leads to an evaluation error if *i* is equal to zero.
- **Boolean < (Integer i)**
  True if the *operand* is less than *i*.
- **Boolean > (Integer i)**
  True if the *operand* is greater than *i*.
- **Boolean <= (Integer i)**
  True if the *operand* is less than or equal to *i*.
- **Boolean != (Integer i)**
  True if the *operand* is not the same as *i*.
- **Boolean <> (Integer i)**
  True if the *operand* is not the same as *i*.
- **Boolean >= (Integer i)**
  True if the *operand* is greater than or equal to *i*.
- **Boolean == (Integer i)**
  True if the *operand* is the same as *i*.
- **Integer - ()**
  The negative value of the *operand*.
- **Integer abs()**
  The absolute value of the *operand*.

Conversions: `Integer` values can automatically be converted to `Real` values.

3.4.2.3 Boolean

The basic type `Boolean` represents the common true/false values.

- **Boolean == (Boolean a)**
  True if the *operand* is the same as *a*.
- **Boolean <> (Boolean a)**
  True if the *operand* is not the same as *a*.
- **Boolean != (Boolean a)**
  True if the *operand* is not the same as *a*.
- **Boolean or (Boolean b)**
  True if either *self* or *b* is true.
- **Boolean xor (Boolean b)**
  True if either *self* or *b* is true, but not both.
- **Boolean and (Boolean b)**
  True if both *b1* and *b* are true.
- **Boolean not ()**
  True if *self* is false and vice versa.
- **Boolean ! ()**
  True if *self* is false and vice versa.

3.4.2.4 String

The standard type `String` represents strings, which can be ASCII.

- **Integer length ()**
  The number of characters in the *operand*.
- **String + (String s)**
  The concatenation of the *operand* and *s*.
- **String + (Path p)**
  The concatenation of the *operand* and the string representation of path *p*.
- **String substring(Integer s, Integer e)**
  Returns the substring of the *operand* from *s* (inclusive) to *e* (exclusive). *operand* is returned in case of any problem, e.g., positions exceeding the valid index range.
- **String replace(String s, String r)**
  Returns *operand* with all substrings *s* replaced by *r*.
- **String substitute(String s, String r, String p)**
  Returns *operand* with all substrings matching the Java regular expression *r* substituted by *p*.
- **Boolean <> (String s)**
  True if the *operand* is not the same as *s*.
- **Boolean != (String s)**
  True if the *operand* is not the same as *s*.
- **Boolean == (String s)**
  True if the *operand* is the same as *s*.
- **Boolean matches (String r)**

Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-adress:

```
mail.matches([\w]*@[\w]*.[\w]*);
```

- **String toUpperCase()**
  Turns the *operand* into a `String` consisting of upper case characters.
- **String toLowerCase()**
  Turns the *operand* into a `String` consisting of lower case characters.
- **String firstToUpperCase()**
  Turns the first character of *operand* into an upper case character.
- **String firstToLowerCase()**
  Turns the first character of *operand* into a lower case character.
- **Integer toInteger ()**
  Converts the *operand* to an Integer value.
- **Real toReal ()**
  Converts the *operand* to a Real value.

3.4.3 Container Types

This section defines the operations of the collection types. VIL defines one abstract collection type `Collection` and two specific collections, namely `Set` and `Sequence`. All collection types are parameterized by one parameter. Below, 'T' will denote the parameter for the collection types. A concrete collection type is created by substituting a type for the parameter T. So a collection of integers is referred in VIL by `setOf(Integer)`.

In addition, VIL defines the type `Map`, an associative container, which allows to relate keys to values.

3.4.3.1 Collection

`Collection` is the abstract super type of all collections in VIL.

- **Integer size ()**
  The number of elements in the collection *operand.*
- **Boolean includes (T object)**
  True if *object* is an element of *operand*, false otherwise.
- **Boolean excludes (T object)**
  True if *object* is not an element of *operand*, false otherwise.
- **Integer count (T object)**
  The number of times that *object* occurs in the collection *operand*.
- **Boolean ==(Collection<Type> c)**
  Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence.
- **Boolean equals(Collection<Type> c)**
  Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence.

- **Boolean isEmpty ()**
  Is the *operand* the empty collection?
- **Boolean isNotEmpty ()**
  Is the *operand* not the empty collection?

3.4.3.2 Set

The type `Set` represents the mathematical concept of a set. It contains elements without duplicates. Set inherits the operations from `Collection`.

- **Sequence<T> toSequence ()**
  Turns *operand* into a sequence.
- **T projectSingle()**
  In case of an *operand* with one element, return that element. Otherwise, nothing will be returned and subsequent expressions may fail.
- **Set<Type> selectByType (Type t)**
  Returns all those elements of *operand* that are type compliant to *t*.
- **Set<T> excluding (Collection<T> s)**
  Returns a subset of *operand*, which does not include the elements in *s*.
- **Set<T> select (Expression e)**
  Returns the elements in *operand*, which comply with the iterator expression *e* (via the implicit iterator variable `ITER`).

3.4.3.3 Sequence

A `Sequence` is a `Collection` in which the elements are ordered. An element may be part of a `Sequence` more than once. `Sequence` inherits the operations from `Collection`.

- **T get (Integer index)**
  Returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **T [] (Integer index)**
  The []-operator returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **T first()**
  Returns the first element of *operand*.
- **T last()**
  Returns the last element of *operand*.
- **Boolean ==(Collection<Type> c)**
  Returns whether *operand* contains the same elements in the same sequence than *c*.
- **Boolean equals(Collection<Type> c)**
  Returns whether *operand* contains the same elements in the same sequence than *c*.
- **Set<T> toSet ()**
  Turns *operand* into a set (excluding duplicates).
- **Sequence<T> selectByType (Type t)**
  Returns all those elements of *operand* that are type compliant to *t*.

- **Sequence <T> excluding (Collection<T> s)**
  Returns a subset of *operand*, which does not include the elements in *s*.
- **Sequence <T> select (Expression e)**
  Returns the elements in *operand* which comply with the iterator expression *e* (via the implicit iterator variable `ITER`).

3.4.3.4 Map

The `Map` type represents an associative container, which is parameterized over the type of keys `K` and the type of values `V`.

- **V get (K key)**
  Returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **V [] (K key)**
  The []-operator returns the value assigned to the given *key* in *operand*.

Sequences which contain sequences with exactly two entry types matching the types of `Map` can be converted automatically into a `Map` instance.

3.4.4 Configuration Types

Configuration types realize the integration with IVML. As the VIL languages are intended for variability instantiation rather than variability modelling, the Configuration types neither support changing the underlying IVML model nor its configuration.

3.4.4.1 IvmlElement

The `IvmlElement` is the most common configuration type. All configuration types discussed in this section are subclasses of `IvmlElement`. Further, this type represents the IVML identifiers used in VIL scripts or templates.

- **Boolean ==(IvmlElement i)**
  Returns whether *operand* is the same IvmlElement as *i*.
- **String getName () / String name ()**
  Returns the (unqualified) name of the *operand*.
- **String getQualifiedName () / String qualifiedName ()**
  Returns the unqualified name of the *operand*.
- **String getType () / String type ()**
  Returns the (unqualified) name of the type of the *operand*.
- **String getQualifiedType () / String qualifiedType ()**
  Returns the unqualified name of the type of the IVML element.
- **IvmlElement getElement(String n) / IvmlElement element (String n)**
  Returns the (nested) element of the *operand* with given name *n*.
- **Attribute getAttribute (String n) / Attribute attribute (String n)**
  Returns the attribute of the *operand* with given name *n*.
- **AnyType getValue () / AnyType value ()**
  Returns the (untyped) configuration value of the *operand*.
- **String getStringValue () / String stringValue ()**

Returns the configuration value of the *operand* as a `String` in case that the underlying IVML decision variable is of type `String` or the `String` representation of the value in any other case.

- **Boolean getBooleanValue () / Boolean booleanValue ()**
  Returns the configuration value of the *operand* as a `Boolean`, but is undefined if the underlying IVML decision variable is not of type `Boolean`.
- **Integer getIntegerValue () / Integer integerValue ()**
  Returns the configuration value of the *operand* as an `Integer`, but is undefined if the underlying IVML decision variable is not of type `Integer`.
- **Real getRealValue () / Real realValue ()**
  Returns the configuration value of the *operand* as a `Real`, but is undefined if the underlying IVML decision variable is not of type `Real`.
- **EnumValue getEnumValue () / Enum enumValue ()**
  Returns the configuration value of the *operand* as an `EnumValue`, but is undefined if the underlying IVML decision variable is not of type `Enum`.

An `IvmlElement` can automatically be converted to a `String` containing the name of the `IvmlElement`.

3.4.4.2 EnumValue

This subtype of `IvmlElement` represents an IVML enumeration value. This subtype of `IvmlElement` does not specify any additional operations.

An `EnumValue` can automatically be converted to a `String` containing the (qualified) name of the `EnumValue`.

3.4.4.3 DecisionVariable

This subtype of `IvmlElement` represents a configured IVML decision variable.

- **Sequence<DecisionVariable> variables()**
  Returns the frozen nested variables of *operand*. Except for the IVML types container and compound, this sequence will always be empty.
- **DecisionVariable getByName(String name) / DecisionVariable byName(String name)**
  Returns the specified decision variable (if it exists).
- **Sequence<Attribute> attributes()**
  Returns the frozen attributes of *operand*. Except for the IVML types container and compound, this sequence will always be empty.
- **Configuration selectAll()**
  Returns a configuration as a projection of the nested decision variables in *operand* also returned by `variables()` but in terms of a configuration.

A `DecisionVariable` can automatically be converted into `Integer`, `Real`, `Boolean`, `String` or `EnumValue` in terms of its respective value and type.

3.4.4.4 Attribute

This subtype of `IvmlElement` represents a configured IVML attribute. This subtype of IvmlElement does not specify any additional operations.

An `Attribute` can automatically be converted into `Integer`, `Real`, `Boolean`, `String` or `EnumValue` in terms of its respective value and type.

3.4.4.5 IvmlDeclaration

This subtype of `IvmlElement` represents the type underlying an IVML decision variable. Instances of this type do not provide any configuration values rather than providing access to the structure of the represented type, e.g., the nested variable declarations in an IVML compound.

3.4.4.6 Configuration

The `Configuration` type provides access to the *frozen* configuration values as well as to the type declarations for a certain IVML model. In particular, the configuration type allows to create projections of a configuration, e.g., to select a subset of decision variables and specify further operations on that subset such as passing it to rules or to (one or more) instantiators. Although being an `IvmlElement`, a configuration instance will not provide access to values.

- **Sequence<DecisionVariable> variables()**
  Returns the configured and frozen decision variables of *operand*.
- **Sequence<Attribute> attributes()**
  Returns the configured and frozen attributes of *operand*.
- **Boolean isEmpty()**
  Returns whether configuration in *operand* is empty, i.e., does not contain decision variables. This may occur due to the projection capabilities of this type.
- **DecisionVariable getByName(String name) / DecisionVariable byName(String name)**
  Returns the specified decision variable (if it exists).
- **Configuration selectByName(String namePattern)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression applied on (qualified and unqualified) decision variable names.
- **Configuration selectByType(String typePattern)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression applied on (qualified and unqualified) type names.
- **Configuration selectByAttribute(String namePattern)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are attributed by the attribute specified in terms of a Java regular expression applied on attribute names.
- **Configuration selectByAttribute(String name, AnyType value)**
  Returns a configuration as a projection of *operand* containing those decision variables which are attributed by the specified attribute (in terms of an IVML identifier) and value.

3.4.5 Built-in Artefact Types and Artefact-related Types

In this section, we will discuss the built-in artefact types. Please note that the (meta model) of the artefact model is extensible, so that further as well as derived types may be added if needed.

3.4.5.1 Path

A path represents a relative or absolute file or folder. Paths are always relative to the containing project, in more detail to the containing artefact model and normalized in Unix notation (using the slash as path separator). Further, paths may be patterns and contain wildcards according to the ANT conventions [9].

- **String getPath() / String path()**
  Returns a string representation of *operand*.
- **Boolean isPattern()**
  Returns whether the *operand* is a pattern, i.e., whether it contains wildcards.
- **JavaPath toJavaPath()**
  Explicitly converts the *operand* into a Java package path.
- **JavaPath toJavaPath(String p)**
  Explicitly converts the *operand* into a Java package path and removes the prefix Java regular expression *p* (might be due to src folders in Eclipse or Maven).
- **String toOSPath()**
  Turns the *operand* into a relative operating system specific path.
- **String toOSAbsolutePath()**
  Turns the *operand* into an absolute operating system specific path.
- **deleteAll()**
  Deletes all artefacts in the *operand* path.
- **mkdir()**
  Creates a directory from the *operand* path. This operation will fail if applied to a pattern.
- **Set<FileSystemArtifact> copy (FileSystemArtifact t)**
  Copies all file system artefacts denoted by the *operand* o the location of *t* and returns the created artefacts at the target location.
- **Set<FileSystemArtifact> move (FileSystemArtifact t)**
  Moves all file system artefacts denoted by the *operand* o the location of *t* and returns the created artefacts at the target location.
- **Boolean matches(Path p)**
  Returns whether the (pattern in) *operand* matches the given path.
- **Set<FileArtifact> selectByType(Type t)**
  Returns those artefacts matching *operand* and complying to the given artefact type *t*.
- **Set<FileArtifact> selectAll()**
  Returns all artefacts matching *operand*.
- **String +(String s)**
  Returns the string concatenation of *operand* and the given String *s*.
- **Boolean exists()**

Returns whether the artefact denoted by the path exists. The operation will always return false in case of a pattern path.

- **delete()**
  Deletes the underlying artefact. This operation will cause no effects on pattern paths.
- **String getName() / String name()**
  Returns the name of the file part of the path or, in case of a pattern path, the entire pattern path.
- **Path rename(String name)**
  Renames the underlying artefact and returns the related new path.

Paths can be constructed from a `String` using the explicit constructor. Typically, scripts shall rely on the automatic conversions from `String` to `Path` or from `Path` to `FileSystemArtifact`.

Conversions: A `Path` can be converted automatically into a `FolderArtifact`.

3.4.5.2 JavaPath

A subtype of `Path` representing Java package paths (separated by ".").

3.4.5.3 Project

The project type encapsulates the physical location of a product line including all artefacts, in particular in terms of an Eclipse project. There are no explicit constructors for this type, as instances will be provided by the VIL/EASy-Producer runtime environment.

- **String getName() / String name()**
  Returns the name of the project in *operand*.
- **String getPlainName() / String plainName()**
  The name of *operand* without file name extension.
- **String getPathSegments() / String pathSegments()**
  The path segments of *operand* without the file name.
- **Set<FileArtifact> selectAllFiles()**
  Returns all file artefacts in *operand*.
- **Set<FileArtifact> selectAllFolders()**
  Returns all folder artefacts in *operand*.
- **Set<Project> predecessors()**
  Returns all predecessor projects in *operand*. In standalone product lines, the result may be empty. In hierarchical product lines, the predecessors are returned.
- **Path getPath() / Path path()**
  Returns the (base) path the *operand* is located in.
- **Path localize(Project s, Path p)**
  Localizes path *p* originally in project s to the project in *operand*.
- **Path localize(Project s, FileSystemArtifact a)**
  Localizes path of *a* originally in project s to the project in *operand*. This operation does neither copy nor move *a*.
- **Set<FileArtifact> selectByType(Type t)**

Returns those file artefacts in the *operand* project which comply with the given type *t*.

- **Set<FileArtifact> selectByName(String r)**
  Returns those file artefacts in the *operand* project for which their name compiles with the Java regular expression in *r*.
- **Path getEasyFolder() / Path easyFolder()**
  Returns the path to the EASy producer configuration files in *operand*.
- **Path getIvmlFolder() / Path ivmlFolder()**
  Returns the path to the IVML models in *operand* (typically the same as `getEasyFolder()`).
- **Path getVilFolder() / Path vilFolder()**
  Returns the path to the VIL build specification models in *operand*.
- **Path getVtlFolder() /Path vtlFolder()**
  Returns the path to the VTL template models in *operand*.

Conversions: A `Project` can be converted automatically into its base `Path`.

3.4.5.4 Text

Represents an artefact or a fragment artefact in terms of the underlying text and allows direct manipulations. The manipulations will be written back into the artefact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artefact.

- **Boolean isEmpty()**
  Returns whether the text representation in *operand* is empty.
- **Boolean matches(String regEx)**
  Returns whether the specified *regEx* matches the textual representation in *operand*.
- **Text substitute(String regEx, String r)**
  Substitutes all occurrences of *regEx* in *operand* by *r* and returns the modified text.
- **Text replace(String s, String r)**
  Substitutes all occurrences of *s* in *operand* by *r* and returns the modified text. This operation does not consider regular expression matches rather than direct text matches.
- **Text append(String s)**
  Appends *s* to the end of *operand* and returns the modified text.
- **Text prepend(String s)**
  Prepends *s* before the beginning of *operand* and returns the modified text.
- **Text append(Text t)**
  Appends the entire contents of *t* to the end of *operand* and returns the modified text.
- **Text prepend(Text s)**
  Prepends the entire contents of *t* before the beginning of *operand* and returns the modified text.

Conversions: Please note that a text representation cannot be converted automatically into a String. This shall avoid confusion as, in contrast to a text

representation, the resulting String is disconnected from the underlying artefact and changes to the String will not affect the artifact.

3.4.5.5 Binary

Represents an artefact or a fragment artefact in terms of the underlying binary form and allows direct manipulations. This type is subject to future extensions. The manipulations will be written back into the artefact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artefact.

- **Boolean isEmpty ()**
  Returns whether the text representation in *operand* is empty.

3.4.5.6 Artifact

The most common artefact type. All specific artefact types are subtypes of `Artifact`.

- **delete()**
  Delete the artefact in *operand* regardless of whether it is a file, a component, or a fragment within an artefact.
- **Boolean exists ()**
  Returns whether the artifact in *operand* actually exists. Please note that the semantics of this operation depends on the type of artefact, e.g., for a FileArtifact this operation returns whether there is an actual underlying file.
- **String getName () / String name()**
  Returns the name of the artefact in *operand*. The specific meaning of the name depends on the actual artefact type.
- **Text getText() / Text text()**
  Returns the textual representation of the artefact in *operand*. Whether the representation can be manipulated depends on whether the artefact itself may be modified.
- **Binary getBinary() / Binary binary()**
  Returns the binary representation of the artefact in *operand*. Whether the representation can be manipulated depends on whether the artefact itself may be modified.
- **rename(String n)**
  Renames the artifact in *operand*. The specific effect of this operation and whether it may be applied at all depends on the actual artefact type.

3.4.5.7 FileSystemArtifact

Represents the most common type of file system artefacts.

- **Path getPath() / Path path()**
  Returns the path to *operand*.
- **Set<FileSystemArtifact> move(FileSystemArtifact a)**
  Moves the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.

- **Set<FileSystemArtifact> copy(FileSystemArtifact a)**
  Copies the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.

Conversions: A `FileSystemArtifact` can be converted automatically into its `Path`.

3.4.5.8 FolderArtifact

This type represents a folder in the file system and always belongs to a certain artifact model (and typically to a containing `Project`). `FolderArtifact` is a subtype of `FileSystemArtifact`.

- **Set<FileArtifact> selectAll ()**
  Returns all file system artifacts contained in *operand*.

Conversions: `FolderArtifact` can automatically be converted into a `String` containing the path or into a `Path` denoting the location.

3.4.5.9 FileArtifact

This type represents a file in the file system and always belongs to a certain artefact model (and typically to a containing `Project`). `FileArtifact` is a subtype of `FileSystemArtifact`. Please note that the actual instance in a variable of type `FileArtifact` may belong to a subtype as the creation of artefact takes artefact specific rules into account.

A temporary `FileArtifact` can be constructed using the constructor without arguments. A string (containing a path) as well as a `Path` can be converted automatically into a `FileArtifact`.

A `FileArtifact` is created as the default fallback, i.e., if no more specific artefact matches the underlying real artefact. The artefact creation mechanism may be configured using an external Java properties file, which relates artefact names to file path patterns (overwriting or extending the built-in rules).

3.4.5.10 VtlFileArtifact

The `VtlFileArtifact` type is a subtype of `FileArtifact`. In particular, it helps the VTL template instantiator in dynamic dispatch between other types of file artefacts and actual VTL templates. It does not provide additional operations or conversions over the `FileArtifact`.

A `VtlFileArtifact` is created for all real file artefacts with file extension `vtl`.

3.4.5.11 XmlFileArtifact

The `XmlFileArtifact` is a built-in composite artefact, i.e., if it exists its content is analysed for known substructures, which are made available for querying and manipulation in terms of fragment artefacts.

- **XmlFileArtifact XmlFileArtifact()**
  Creates a temporary XML file artefact.
- **XmlElement getRootElement() / XmlElement rootElement()**

Returns the root element of the XML artifact in *operand*.

- **Set<XmlElement> selectAll()**
  Returns all XML elements contained in *operand*.
- **Set<XmlElement> selectChilds()**
  Returns all XML elements contained in the root element of *operand*.
- **Set<XmlAttribute> selectByName (String r)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression.
- **Set<XmlAttribute> selectByPath (String p)**
  Returns those XML elements in *operand*, which comply with the given name path *p*. Here, a path is a hierarchical name separated by slashes denoting the names of nested elements (except for the root element which is implicitly selected). Path elements may be Java regular (sub-)expressions.
- **Set<XmlAttribute> selectByRegEx (String r)**
  Returns those XML elements in *operand*, which comply regarding their name with the given Java regular expression *r*.

A `XmlFileArtifact` is created for all real file artefacts with file extension `xml`.

XmlElement

The `XmlElement` is a built-in fragment artefact, which belongs to the `XmlFileArtifact`. In particular, it inherits all operations from `Artifact` such as access to the representations of the contained text or CDATA.

- **XmlElement XmlElement(XmlElement p, String n)**
  Creates a new XmlElement with name *n* as child of the parent element *p*.
- **XmlElement XmlElement(XmlFileArtifact p, String n)**
  Creates a new XmlElement with name *n* as child of the parent XMLFileArtifact *p*. This is just a convenience constructor which actually executes `XMLElement(p.getRootElement(), n)`.
- **Set<XmlElement> selectAll()**
  Returns all XML elements contained in *operand*.
- **Set<XmlAttributes> attributes()**
  Returns all XML attributes belonging to *operand*.
- **Text getCdata()**
  Returns the CDATA information of *operand* as a textual representation.
- **XmlAttributes getAttribute(String n)**
  Returns the XML attribute in *operand* with the specified name *n*. Please note that this operation does not fail, if the specified attribute does not exist but rather the subsequent evaluation will stop.
- **XmlAttributes setAttribute(String n, String v)**
  Defines or changes the XML attribute in *operand* with the specified name *n* to the given value *v*.
- **Set<XmlAttribute> selectByName (String r)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression.

- **Set<XmlAttribute> selectByName (String r, Boolean c)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression. The parameter *c* denotes whether the name comparison shall consider case sensitivity.
- **Set<XmlAttribute> selectByRegEx (String r)**
  Returns those XML elements in *operand*, which comply regarding their name with the given Java regular expression *r*.
- **Set<XmlAttribute> selectByPath (String path)**
  Returns those XML elements in *operand*, which comply with the given name path. Here, a path is a hierarchical name separated by slashes denoting the names of nested elements (except for the root element which is implicitly selected). Path elements may be Java regular (sub-)expressions.

XmlAttribute

The `XmlElement` is a built-in fragment artefact, which belongs to the `XmlFileArtifact` and to the fragment artefact `XmlElement`. In particular, it inherits all operations from `Artifact`.

- **XmlAttribute XmlAttribute(XmlElement p, String n, String v, Boolean f)**
  Creates a new XML attribute for in *p* with name *n* and value *v*. In case that an equally named attribute already exists in p, the attribute is overwritten if *f* is true or created anyway (*f* is false).
- **XmlAttribute XmlAttribute(XmlElement p, String n, String v)**
  Creates a new XML attribute for in *p* with name *n* and value *v* by executing `XmlAttribute(p, n, v, true)`.
- **String getValue () / String value()**
  Returns the value of the attribute in *operand*.
- **setValue (String v)**
  Changes the value of the attribute in *operand* to *v*.

3.4.5.12 JavaFileArtifact

The `JavaFileArtifact`[11] is a built-in composite artefact and allows querying fragment artefacts as well as and (simple) manipulation of Java source code artefacts[12]. Please note that the `JavaFileArtifact` represents a Java compilation unit. In addition to the file artefact operations, this artefact defines the following operations:

- **Set<JavaClass> classes()**
  Returns all classes defined by the *operand* artefact.
- **renamePackages(String o, String n)**[13]

[11] Please not that the Java source code part of VIL / VTL is still under development, i.e., functionality may change / be added in the future.

[12] There is also a `JavaFileArtifact`, which is actually resolved by VIL and VTL but it is currently not implemented and kept due to existing test cases.

[13] Renaming of packages is currently under realization.

Renames all packages in *operand* from *o* to n. Nothing happens, if *o* cannot be found.

- **renamePackages(Map m)**[13]
  Renames all packages in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.
- **Boolean hasAnnotation (String a, String f, String v)**
  Determines whether one of the classes, methods or fields in the *operand* defines the specified annotation named *a* with field *f* and value *v*. The annotation name *a* may be given as qualified or unqualified name[14]. The field name *f* may be given as empty string (then the default name "value" is used).

A `JavaFileArtifact` is created for all real file artefacts with file extension `java`.

JavaClass

The `JavaClass` is a built-in fragment artefact, which belongs to the `JavaFileArtifact`. It represents a single class within a Java compilation unit. The `JavaClass` defines the following operations:

- **Set<JavaAnnotation> annotations()**
  Returns all annotations defined by the *operand*.
- **Set<JavaAttribute> attribute()**
  Returns all attributes defined by the *operand.*
- **Set<JavaMethod> methods()**
  Returns all methods defined by the *operand.*
- **Set<JavaMethod> classes()**
  Returns all classes defined by the *operand* artefact.

JavaAnnotation

The `JavaAnnotation` is a built-in fragment artefact, which represents a Java annotation attached to a class, a method or an attribute. The `JavaAnnotation` defines the following operations:

- **String getQualifiedName() / String qualifiedName()**
  Returns the qualified name14 of the *operand*. **getName() / name()** returns the simple name.
- **SetOf<String> fields ()**
  Returns the fields of the *operand*.
- **String getAnnotationValue(String f)**
  Returns the value of the annotation for field *f* of the *operand***.** If the field *f* does not exist, VIL / VTL will ignore the related expression.

[14] Please note that we parse Java files but do not resolve them so that qualified names are only available if they are explicitly stated in the source code.

JavaMethod

The `JavaMethod` is a built-in fragment artefact, which represents a Java method as part of a class. The `JavaMethod` defines the following operations:

- **String getQualifiedName() / String qualifiedName()**
  Returns the qualified name14 of the *operand*. **getName() / name()** returns the simple name.
- **Set<JavaAnnotation> annotations()**
  Returns all annotations defined by the *operand*.

JavaAttribute

The `JavaAttribute` is a built-in fragment artefact, which represents a Java attribute (field) as part of a class. The `JavaAttribute` defines the following operations:

- **String getQualifiedName() / String qualifiedName()**
  Returns the qualified name14 of the *operand*. **getName() / name()** returns the simple name.
- **setValue(Any value)**
  Defines the (initial) value of the operand. The value will be emitted as part of the attribute declaration.
- **makeConstant()**
  Turns the *operand* into a “constant” (if it is not already a “constant”), i.e., makes it static final.
- **makeVariable()**
  Turns the *operand* into a “variable” (if it is not already a “variable”), i.e., removes static final.
- **Set<JavaAnnotation> annotations()**
  Returns all annotations defined by the *operand*.

3.4.6 Built-in Instantiators

VIL provides also several built-in instantiators, in particular to modify or generate entire artefacts in one step. Basically, instantiators shall be defined using the VIL template language (this actually happens through an instantiator for VIL templates). However, very complex instantiation operations as well as existing (legacy) instantiator operations shall be available to the VIL build language, also as a better integrated alternative to just calling an operating system command. In this section, we will discuss the instantiators shipped with the VIL implementation as well as their specific operations. Please refer to the EASy-Producer developer documentation on how to realize custom instantiators.

3.4.6.1 VIL Template Processor

The VIL template processor is responsible for interpreting and executing VIL template scripts in close collaboration with the VIL build language. It may operate in two different modes depending on the actual arguments, namely executing VIL templates or replacing VIL expressions in a file artefact.

Basically, VIL templates receive three different parameters, the template, the variability configuration and the target artefact (fragment) to be produced. Thereby, the instantiator itself takes an argument of type `Artifact`, but the dynamic dispatch mechanism allows specifying subtypes in the template parameters or even to have multiple main subtemplates for different artefact types. In addition the VIL template processor may receive an arbitrary number of named arguments specific to the template to be executed.

This instantiator provides three instantiator operations:

- **Set<FileArtifact> vilTemplateProcessor(String n, Configuration c, Artifact a, …)**
- Parses and analyses the template given by its name *n* (version restriction see Section 3.1.4), executes the template specification using the configuration *c* and replaces the content of *a*. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **Set<FileArtifact> vilTemplateProcessor(VtlFileArtifact t, Configuration c, Artifact a, …)**
  Parses and analyses the template in *t*, executes the template specification using the configuration *c* and replaces the content of *a*. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **Set<FileArtifact> vilTemplateProcessor(FileArtifact t, Configuration c, Artifact a)**
  Searches for VIL variables and expressions in *t* (variables given as `$variableName`, expressions as `${expression}`) and replaces them with their actual value as defined in the configuration *c*. The output replaces the content of *a*.

In addition, similar operations are provided which take a **collection of artefacts** as input. At a glance, storing the output produced by the same template in multiple files might be superfluous but it simplifies the application of the VTL template processor in conjunction with operations which return a collection with one element, such as copying a file (without further utilizing the `projectSingle` operation).

3.4.6.2 Blackbox Instantiators

In this section, we describe three built-in blackbox instantiators.

Velocity Instantiator

The Velocity[15] instantiator [4] allows using one of the basic functionalities of EASy through VIL. It provides instantiator calls for individual templates and collections of templates.

- **Set<FileArtifact> velocity(FileArtifact t, Configuration c)**
  Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.

[15] http://velocity.apache.org/engine/devel/user-guide.html#Velocity_Template_Language_VTL:_An_Introduction

- **Set<FileArtifact> velocity(Collection<FileArtifact> t, Configuration c)**
  Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.
- **Set<FileArtifact> velocity(Path t, Configuration c, Map m)**
  Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result into *t*. Thereby, the variable names in *c* are replaced for the template processing by the name-name mapping in *m*, e.g., to enable a mapping of variabilities to implementation names on this level.
- **Set<FileArtifact> velocity(Collection<FileArtifact> t, Configuration c, Map m)**
  Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result into *t*. Thereby, the variable names in *c* are replaced for the template processing by the name-name mapping in *m*, e.g., to enable a mapping of variabilities to implementation names on this level.

Java Compiler

The Java compiler blackbox instantiator allows to directly compile Java source code artefacts from the VIL build language. It provides the following instantiator call

- **Set<FileArtifact> javac(Path s, Path t, ...)**
  Compiles the artefacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artefacts denoting the classpath, for example
  sequenceOf(String) cp = {"$source/lib/myLib.jar"};
  Javac("$source/**/*.java", "$target/bin", classpath=cp);
- **Set<FileArtifact> javac(Collection<FileArtifact> s, Path t, ...)**
  Compiles the artefacts denoted by *s* into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artefacts denoting the classpath.

AspectJ Compiler

The AspectJ [1] compiler blackbox instantiator allows to directly compile Java and AspectJ source artefacts from the VIL build language. It provides the following instantiator call

- **Set<FileArtifact> aspectJ(Path s, Path t, ...)**
  Compiles the artefacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.
- **Set<FileArtifact> aspectJ(Collection<FileArtifact> s, Path t, ...)**
  Compiles the artefacts denoted by *s* into the target path *t*. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.

Zip File Instantiator

The ZIP file blackbox instantiator allows to pack and unpack ZIP files.

- **Set<FileArtifact> zip(Path b, Path a, Path z)**
  Packs the artefacts denoted by the path *a* (possibly a path pattern) into the ZIP file denoted by path *z* while taking path *b* (or a project) as a basis for making the file names of the artefacts relative in the resulting ZIP file. If *z* does not exist, a new artefact is created. If *z* exists, the contents of *z* is taken over into the result ZIP, i.e., the artefacts in *a* are added to *z*.
- **Set<FileArtifact> zip(Path b, Collection<FileArtifact> a, Path z)**
  Packs the artefacts in *a* into the ZIP file denoted by path *z* while taking path *b* (or a project) as a basis for making the file names of the artefacts relative in the resulting ZIP file. If *z* does not exist, a new artefact is created. If *z* exists, the contents of *z* is taken over into the result ZIP, i.e., the artefacts in *a* are added to *z*.
- **Set<FileArtifact> unzip(Path z, Path t)**
  Unpacks artefacts in the ZIP file *z* into the target path *t*.
- **Set<FileArtifact> unzip(Path z, Path t, String p)**
  Unpacks those artefacts in the ZIP file *z* matching the ANT pattern *p* into the target path *t*.

JAR File Instantiator

The JAR (Java Archive) file blackbox instantiator allows to pack and unpack Jar files.

- **Set<FileArtifact> jar(Path b, Path a, Path j)**
  Packs the artefacts denoted by the path *a* (possibly a path pattern) into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artefacts relative in the resulting JAR file. If *j* does not exist, a new artefact is created with an empty manifest. If *j* exists, the contents of *j* (including the manifest) is taken over into the result JAR, i.e., the artefacts in *a* are added to *j*.
- **Set<FileArtifact> jar(Path b, Collection<FileArtifact> a, Path j)**
  Packs the artefacts in *a* into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artefacts relative in the resulting JAR file. If *j* does not exist, a new artefact is created with an empty manifest. If *j* exists, the contents of *j* (including the manifest) is taken over into the result JAR, i.e., the artefacts in *a* are added to *j*.
- **Set<FileArtifact> jar(Path b, Path a, Path j, Path m)**
  Packs the artefacts denoted by the path *a* (possibly a path pattern) into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artefacts relative in the resulting JAR file. If *j* does not exist, a new artefact is created and *m* will be the manifest. If *j* exists, the contents of *j* is taken over into the result JAR, i.e., the artefacts in *a* are added to *j* while the manifest *m* will take precedence over the existing manifest.
- **Set<FileArtifact> jar(Path b, Collection<FileArtifact> a, Path j, Path m)**
  Packs the artefacts in *a* into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artefacts relative in the resulting JAR file. If *j* does not exist, a new artefact is created. If *j* exists,

the contents of *j* is taken over into the result JAR, i.e., the artefacts in *a* are added to *j* while the manifest *m* will take precedence over the existing manifest.

- **Set<FileArtifact> unjar(Path j, Path t)**
  Unpacks artefacts in the JAR file *j* into the target path *t*.
- **Set<FileArtifact> unjar(Path j, Path t, String p)**
  Unpacks those artefacts in the JAR file *j* matching the ANT pattern *p* into the target path *t*.
- **Set<FileArtifact> unjar(Path j, Path t, Boolean m)**
  Unpacks artefacts in the JAR file *j* into the target path *t* whereby *m* determines whether the manifest shall also be unpacked.
- **Set<FileArtifact> unjar(Path j, Path t, String p, Boolean m)**
  Unpacks those artefacts in the JAR file *j* matching the ANT pattern *p* into the target path *t* whereby *m* determines whether the manifest shall also be unpacked.

4 How to ...?

Learning a new language is frequently simplified if examples are provided. This is in particular true for languages which include a rich library of operations. In addition to the illustrating examples shown in the sections above, we will discuss a collection of typical application patterns in this section. In particular, this section is meant to be a live document, i.e., this section will be extended over time and is not intended to be comprehensive at the moment.

4.1 VIL Build Language

4.1.1 Copy Multiple Files

One recurring task is to copy multiple files, frequently from the source to the target project in order to prepare instantiation. Basically, multiple files can be described by a regular path expression in ANT notation. Let's assume that we want to copy all Java files in the `src` folder of the source project to the `src` folder of the target project:

The hard way

Copying all those files can be described in imperative fashion as follows:

```
Path p = "$source/src/**/*.java";
map(f = p.selectAll()) {
  copy(f, "$target" + p.path().substring("$source".length());
}
```

However, this needs abusing a map as a loop and manually taking care of the target artefact names and it does not care whether files actually need to be copied.

The path copy operation

In order to simplify the copy fragment above, we can just use the related path operation:

```
Path p = "$source/src/**/*.java";
copy(p, $target); // or p.copy($target) if you like
```

Although this is much simpler, it still copies all files.

A bit smarter

Instead of imperative coding, we rely on VIL rules:

```
doCopy() = "$target/src/**/*.java" : "$source/src/**/*.java" {
  copy(RHS, LHS);
}
```

In this fragment, the VIL pattern matching algorithm takes care of relating source and target artefacts and copies only files if needed, i.e., the target does not exist or is outdated. However, for each pair of patterns you need a specific rule.

Smart and reusable

To make the copy rule shown above reusable, we introduce a parameter:

```
doCopy(String base) = "$target/$base/**/*.java" :
  "$source/$base/**/*.java" {
  copy(RHS, LHS);
}
```

4.1.2 Convenient Shortcuts

Sometimes selection or artefact operations lead to exactly one element as it is intended by the script designer due to the structure of the variability model or the realization of the product line. However, these operations return a collection instead of a single value so that `sequence[0]`, `sequence.get(0)`, `sequence.first()`, `set.toSequence().get(0)` etc. must be applied to obtain that single instance.

In case of sets, this can be simplified by `set.projectSingle()`. Further, if the instance shall further be processed by an instantiator or the VIL template processor, frequently also a collection can be passed in instead of a single artefact so that the operations shown above are not required at all, e.g.,

```
vilTemplateProcessor(”template.vtl”, config, set);
```

4.1.3 Projected Configurations

Frequently, templates or the velocity instantiator do not need to access to the full configuration. Basically, passing in subsets of a configuration speeds up the instantiation process. However, in some cases, it is even required to work on a subset of the configuration, e.g., to select a certain element of an IVML container and to continue on the decision variables of that element, e.g., a compound. Consider the following IVML fragment

```
Compound Workflow {
  String name;
  Boolean enabled;
}
sequenceOf(Workflow) workflows;
```

Here, the user may configure different workflows the resulting product shall provide. While generating the workflow implementation (bindings) with VIL or their configuration, typically each configured workflow is processed and the values are taken over. If this shall be realized with velocity, the velocity processor does not know which workflow actually shall be processed. A simple solution in VIL is

```
map(wf = config.selectByName(”workflows”).variables) {
  String wfName = wf.selectByName(”name”).stringValue();
  copy(wfTemplate, ”$target/src/workflows/$wfName.java”);
  velocity(wfFile, wf.selectAll());
}
```

This fragment processes all compound instances in workflows, prepares the instantiation of each workflow by copying the workflow template `wfTemplate` to the right location and by finally by instantiating the template using velocity. Thereby, the nested decision variables from the actual workflow `wf` are projected into a configuration (`wf.selectAll()`) and passed to velocity where then $name and $enbled can directly be used as placeholders for the actual values.

4.2 VIL Template Language

In this section we will discuss some patterns for the VIL template language.

4.2.1 Don't fear named parameters

Basically, a VIL template takes two parameters, the configuration and the target artefact. In many situations, already further parameter are helpful, just to parameterize the template or to pass already determined information (from artifacts, the configuration or both) into the template and to simplify the template. Therefore, a VIL template may take an arbitrary number of named parameters.

```
template properties(Configuration config, FileArtifact target,
  String name) {

  def main(Configuration config, FileArtifact target, String
  name) {
   //...
  }
}
```

The respective call in IVML would then look like

```
vilTemplateProcessor ("properties.vtl", config, target,
  name="myName");
```

Please note that this works with arbitrary types and that VIL type conversion applies.

4.2.2 Appending or Prepending

While in some situations the complete creation of an artefact is required, in others it is sufficient to append or prepend information to the contents of an artefact.

The Imperative Style

Basically, we may obtain the contents of the artefact in terms of its textual representation and use the provided operations, e.g.,

```
def main(Configuration config, FileArtifact target) {
  Text targetText = target.getText();
  targetText.append("\n");
  targetText.append("Information ${config.name()}\n");
}
```

The modifications to the text representation will be synched into the artefact as soon as the target variable is reclaimed by the runtime environment, i.e., at the end of the subtemplate. The advance of this approach is that it works in the same way in

the VIL build script so that a template may be superfluous. However, stating the required operation in each line and explicitly caring for the line ends is tedious.

Mixing contents

As an alternative, we can simply use the targetText variable within a content statement:

```
def main(Configuration config, FileArtifact target) {
  Text targetText = target.getText();
  '${targetText.text()}

   Information ${config.name()}'
}
```

Please note that a text representation is not automatically converted into a String in order to emphasize that the resulting String is disconnected from the underlying artefact while operations on the text representation will affect the artefact.

4.3 Both languages

In this section, we summarize some patterns applicable to both languages (in order to avoid repetitions).

4.3.1 Rely Automatic Conversions

Retrieving values from a decision variable may become lengthy due to the explicit type access to the actual variable.

```
Boolean value = config.byName(Variability).booleanValue();
```

As a shorter alternative, you may rely on automated conversions, i.e.,

```
Boolean value = config.byName(Variability);
```

5 Implementation Status

The development and realization of VIL and VTL related tools is still in progress. In this section, we summarize the current status.

Missing / incomplete functionality

- Full versioned imports as the basic mechanism taken over from IVML is not working with xText at the moment.
- Affected artifacts in VIL may be incomplete.
- Package renaming in JavaFileArtifact is currently under realization.

6 VIL Grammars

In this section we depict the actual grammar for the VIL languages. The grammar is given in terms of a simplified xText[16] grammar (close to ANTLR[17] or EBNF). Simplified means, that we omitted technical details used in xText to properly generate the underlying EMF model as well as trailing ";" (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

6.1 VIL Build Language Grammar

```
ImplementationUnit:
  Import*
  Require*
  LanguageUnit*;


LanguageUnit:
  Advice* 'vilScript' Identifier '(' ParameterList? ')'
  (ScriptParentDecl)? '{'
    VersionStmt?
    LoadProperties*
    ScriptContents
  '}' ';'?

Require:
  'requireVTL' STRING versionSpec ';'

ScriptParentDecl:
  'extends' Identifier

LoadProperties:
  'load' 'properties' STRING ';'

ScriptContents:
  (VariableDeclaration | RuleDeclaration)*
```

[16] http://www.eclipse.org/Xtext/

[17] http://www.antlr.org

```
RuleDeclaration:
  (RuleModifier? Identifier '(' (ParameterList)? ')' '=')?
  (LogicalExpression)?  ':'
  (LogicalExpression (',' LogicalExpression)*)?
  RuleElementBlock ';'?

RuleElementBlock:
  '{' RuleElement* '}'

RuleElement:
  VariableDeclaration
  | ExpressionStatement
  | DeferDeclaration

RuleModifier:
  'protected'

PrimaryExpression:
  ExpressionOrQualifiedExecution
  | UnqualifiedExecution
  | SuperExecution
  | SystemExecution
  | Map
  | Join
  | Instantiate
  | ConstructorExecution

Map:
  'map' '(' Identifier (',' Identifier)* '=' Expression ')'
  RuleElementBlock ';'?

Join:
  'join' '(' JoinVariable ',' JoinVariable ')'
  ('with' '(' Expression ')')?

JoinVariable:
  'exclude'? Identifier ':' Expression

SystemExecution:
  'execute' Call SubCall*
```

```
Instantiate:
  'instantiate' ((Identifier ('rule' STRING)?) | STRING)
    '(' ArgumentList? ')' VersionSpec?
```

6.2 VIL Template Language Grammar

```
LanguageUnit:
  Import*
  Extension*

  Advice*
  IndentationHint?
  'template' Identifier '(' ParameterList? ')'
  ('extends' Identifier)? '{'
    VersionStmt?
    VariableDeclaration*
    VilDef*
  '}'

IndentationHint:
  '@indent' '(' IndentationHintPart (',' IndentationHintPart)* ')'

IndentationHintPart:
  Identifier '=' NUMBER

VilDef:
  'def' Type? Identifier '(' ParameterList? ')' StmtBlock ';'?

StmtBlock:
    '{' Stmt* '}'

Stmt:
  VariableDeclaration
  | Alternative
  | Switch
  | StmtBlock
  | Loop
  | ExpressionStatement
  | Content
```

```
Alternative:
  'if' '(' Expression ')' Stmt (=> 'else' Stmt)?

Content:
  (STRING) ('|' Expression ';')?
Switch:
  'switch' '(' Expression ')' '{'
    (SwitchPart (',' SwitchPart)* (',' 'default' ':' Expression)?)
  '}'

SwitchPart:
  Expression ':' Expression

Loop:
  'for' '(' Type Identifier ':' Expression
    (',' PrimaryExpression)
      (',' PrimaryExpression)?
     ')'? Stmt

Extension:
  'extension' JavaQualifiedName ';'

JavaQualifiedName:
    Identifier ('.' Identifier)*
```

6.3 Common Expression Language Grammar

Actually, parts of this common language are overridden and redefined by the two VIL language grammars.

```
LanguageUnit:
  Import*
  Advice*
  Identifier
  VersionStmt?

VariableDeclaration:
  'const'? Type Identifier ('=' Expression)? ';'

Advice:
  '@advice' '(' QualifiedName ')' VersionSpec?
```

```
VersionSpec:
  'with' '(' VersionedId  (',' VersionedId)* ')'

VersionedId:
  'version' VersionOperator VERSION


VersionOperator:
  '==' |'>' |'<' |'>=' |'<='

ParameterList:
  (Parameter (',' Parameter)*)

Parameter:
  Type Identifier

VersionStmt:
  'version' VERSION ';'

Import:
  'import' Identifier VersionSpec? ';'

ExpressionStatement:
  (Identifier '=')? Expression ';'

Expression:
  LogicalExpression | ContainerInitializer

LogicalExpression:
  EqualityExpression  LogicalExpressionPart*

LogicalExpressionPart:
  LogicalOperator EqualityExpression

LogicalOperator:
  'and' |'or' |'xor'

EqualityExpression:
  RelationalExpression EqualityExpressionPart?
```

```
EqualityExpressionPart:
  EqualityOperator RelationalExpression

EqualityOperator:
  '==' | '<>' | '!='

RelationalExpression:
  AdditiveExpression RelationalExpressionPart?

RelationalExpressionPart:
  RelationalOperator AdditiveExpression

RelationalOperator:
  '>' | '<' | '>=' | '<='

AdditiveExpression:
  MultiplicativeExpression AdditiveExpressionPart*

AdditiveExpressionPart:
  AdditiveOperator MultiplicativeExpression

AdditiveOperator:
  '+' | '-'

MultiplicativeExpression:
  UnaryExpression MultiplicativeExpressionPart?

MultiplicativeExpressionPart:
  MultiplicativeOperator UnaryExpression

MultiplicativeOperator:
  '*' | '/'

UnaryExpression:
    UnaryOperator? PostfixExpression

UnaryOperator:
  'not' | '!' | '-'
```

```
PostfixExpression:  // for extension
  PrimaryExpression

PrimaryExpression:
  ExpressionOrQualifiedExecution
  | UnqualifiedExecution
  | SuperExecution
  | ConstructorExecution

ExpressionOrQualifiedExecution:
  (Constant | '(' Expression ')') SubCall*

UnqualifiedExecution:
  Call SubCall*

SuperExecution:
  'super' '.' Call SubCall*

ConstructorExecution:
  'new' Type '(' ArgumentList? ')' SubCall*

SubCall:
  '.' Call | '[' Expression ']'

Call:
  QualifiedPrefix '(' param=ArgumentList? ')'

ArgumentList:
  NamedArgument (',' NamedArgument)*

NamedArgument:
  (Identifier '=')? Expression

QualifiedPrefix:
  Identifier ('::' Identifier)*

QualifiedName:
  QualifiedPrefix ('.' Identifier)?
```

```
Constant:
  NumValue | STRING | QualifiedName | ('true' | 'false')


NumValue :
  NUMBER

Identifier:
  ID

Type:
  Identifier
  | ('setOf' TypeParameters)
  | ('sequenceOf' TypeParameters)
  | ('mapOf' TypeParameters)

TypeParameters:
  '(' Identifier (',' Identifier)* ')'

ContainerInitializer:
  '{' (ContainerInitializerExpression
    (',' ContainerInitializerExpression)*)? '}'

ContainerInitializerExpression:
    LogicalExpression |ContainerInitializer

terminal VERSION:
  'v' ('0'..'9')+ ('.' ('0'..'9')+)*

terminal ID:
  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*

terminal NUMBER:
  '-'?
  (('0'..'9')+ ('.' ('0'..'9')* EXPONENT?)?
  | '.' ('0'..'9')+ EXPONENT?
  | ('0'..'9')+ EXPONENT)

terminal EXPONENT:
```

```
  ('e'|'E') ('+'|'-')? ('0'..'9')+

terminal STRING :
  '"'
    ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )*
  '"' | "'"
    ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )*
  "'"

terminal ML_COMMENT:
  '/*' -> '*/'

terminal SL_COMMENT:
  '//' !('\n'|'\r')* ('\r'? '\n')?

terminal WS:
  (' '|'\t'|'\r'|'\n')+

terminal ANY_OTHER:
  .
```

References

[1] Project homepage AspectJ, 2011. Online available at: http://www.eclipse.org/aspectj/.

[2] Eclipse Foundation. Xtend - Modernize Java, 2013. Online available at: http://www.eclipse.org/xtend.

[3] INDENICA Consortium. Deliverable D2.1: Open Variability Modelling Approach for Service Ecosystems. Technical report, 2011.

[4] INDENICA Consortium. Deliverable D2.4.1: Variability Engineering Tool (interim). Technical report, 2012.

[5] INDENICA Consortium. Deliverable D2.2.2: Variability Implementation Techniques for Platforms and Services (final). Technical report, 2013.

[6] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: http://www.omg.org/docs/formal/06-05-01.pdf.

[7] SSE. Ivml language specification. http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf [validated: March 2013].

[8] Richard M. Stallmann, Roland McGrath, and Paul D. Smith. GNU Make - A Program for Directing Recompilation - GNU make Version 3.82, 2010. Online available at: http://www.gnu.org/software/make/manual/make.pdf.

[9] The Apache Software Foundation. Apache Ant 1.8.2 Manual, 2013. Online available at: http://ant.apache.org/manual/index.html.